Doubly-efficient zkSNARKs without trusted setup

Riad S. Wahby[*]
rsw@cs.stanford.edu

Ioanna Tzialla[°]
iontzialla@gmail.com

abhi shelat[†]
abhi@neu.edu

Justin Thaler[‡]
justin.thaler@georgetown.edu

Michael Walfish[°]
mwalfish@cs.nyu.edu

[*]Stanford [°]NYU [†]Northeastern [‡]Georgetown

Abstract. We present a zero-knowledge argument for NP with low communication complexity, low concrete cost for both the prover and the verifier, and no trusted setup, based on standard cryptographic assumptions. Communication is proportional to $d \cdot \log(G)$ (for d the depth and G the width of the verifying circuit) plus the square root of the witness size. When applied to batched or data-parallel statements, the prover's runtime is linear and the verifier's is sub-linear in the verifying circuit size, both with good constants. In addition, witness-related communication can be reduced, at the cost of increased verifier runtime, by leveraging a new commitment scheme for multilinear polynomials, which may be of independent interest. These properties represent a new point in the tradeoffs among setup, complexity assumptions, proof size, and computational cost.

We apply the Fiat-Shamir heuristic to this argument to produce a zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) in the random oracle model, based on the discrete log assumption, which we call Hyrax. We implement Hyrax and evaluate it against five state-of-the-art baseline systems. Our evaluation shows that, even for modest problem sizes, Hyrax gives smaller proofs than all but the most computationally costly baseline, and that its prover and verifier are each faster than three of the five baselines.

1 Introduction

A zero-knowledge proof convinces a verifier of a statement while revealing nothing but its own validity. Since they were introduced by Goldwasser, Micali, and Rackoff [50], zero-knowledge (ZK) proofs have found applications in domains as diverse as authentication and signature schemes [85, 90], secure encryption [40, 89], and emerging blockchain technologies [12].

A seminal result in the theory of interactive proofs and cryptography is that any problem solvable by an interactive proof (IP) is also solvable by a computational zero-knowledge proof or perfect zero-knowledge argument [8]. This means that, given an interactive proof for any NP-complete problem, one can construct zero-knowledge proofs or arguments for any NP statement. But existing instantiations of this paradigm have large overheads: early techniques [22, 48] require many repetitions to achieve negligible soundness error, and incur polynomial blowups in prover work and communication. More recent work [24, 27, 28, 30, 51, 52, 55] avoids those issues, but generally entails many expensive cryptographic operations.[1]

Several other recent lines of work have sought to avoid these overheads. As detailed in Section 2, however, these works still yield costly protocols or come with significant limitations. In particular, state-of-the-art, general-purpose ZK protocols suffer

from one or more of the following problems: (a) they require proof size that is linear or super-linear in the size of the computation verifying an NP witness; (b) they require the prover or verifier to perform work that is super-linear in the time to verify a witness; (c) they require a complex parameter setup to be performed by a trusted party; (d) they rely on non-standard cryptographic assumptions; or (e) they have very high concrete overheads. These issues have limited the use of such general-purpose ZK proof systems in many contexts.

Our goal in this work is to address the limitations of existing general-purpose ZK proofs and arguments. Specifically, we would like to take any computation for verifying an NP statement and turn it into a zero-knowledge proof of the statement's validity. In addition to concrete efficiency, our desiderata are that:

- the proof should be *succinct*, that is, sub-linear in the size of the statement and the witness to the statement's validity;
- the verifier should run in time linear in input plus proof size;
- the prover, given a witness to the statement's validity, should run in time linear in the cost of the NP verification procedure;
- the scheme should not require a trusted setup phase or common reference string; and
- soundness and zero-knowledge should each be either statistical or based on standard cryptographic assumptions. Pragmatically, security in the random oracle model [7] suffices.

Our approach transforms a state-of-the-art interactive proof for arithmetic circuit (AC) satisfiability into a zero-knowledge argument by composing new ideas with existing techniques.

Ben-Or et al. [8] and Cramer and Damgård [37] show how to transform IPs into computationally ZK proofs or perfectly ZK arguments, using cryptographic commitment schemes. At a high level, rather than sending its messages in the clear, the prover sends cryptographic *commitments* corresponding to its messages. These commitments are *binding*, ensuring that the prover cannot cheat by equivocating about its messages. They are also *hiding*, meaning that the verifier cannot learn the committed value and thus ensuring zero-knowledge. Finally, the commitment scheme has a homomorphism property (§3.1) that allows the verifier to check the prover's messages "underneath the commitments."

Accepted wisdom is that such transformations introduce large overheads (e.g., [33, §1.1]). In this paper, we challenge that wisdom by constructing a protocol that meets our desiderata for many cases of interest.

Our starting point is the Giraffe interactive proof [104] with an optimization, adapted from Chiesa et al. [33], that reduces communication complexity (§3.2). We transform this IP into a ZK argument through a straightforward (but careful) application of Cramer-Damgård techniques (§4). This argument uses cryptographic operations (required by the commitment schemes)

---

[1] Some works avoid these overheads by targeting specific problems with algebraic structure and cryptographic significance, most notably Schnorr-style proofs [90] for languages related to statements about discrete logarithms of group elements.

only for the witness and for the prover's messages, which are sub-linear in the size of the AC. (In contrast, many recent works invoke cryptographic primitives for *each gate* in the verifying circuit [12, 13, 16, 24, 30, 46, 79]; §2.) But the argument is not succinct, and it has high concrete costs, especially for the verifier.

We slash these costs with two key refinements. First, we exploit the IP's structure by tightly integrating the verification procedure with a multi-commitment scheme and a Schnorr-style proof [90] (§5); this reduces communication and computational costs by 3–5× compared to the naive approach. Second, we devise a new witness commitment scheme (§6), yielding a succinct argument and asymptotically reducing the verifier's cost associated with the witness.

Our protocol is public coin; we compile it into *Hyrax*, a zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) [20] in the random oracle model [7], via the Fiat-Shamir heuristic [41] (§7). We evaluate Hyrax and five state-of-the-art baselines (BCCGP-sqrt [24], Bulletproofs [30], Ligero [1], ZKB++ [32], and libSTARK [11]; §8). For modest problem sizes, Hyrax gives smaller proofs than all but the most computationally costly baseline; its prover and verifier are each faster than three of the five baselines; and its refinements yield multiple-orders-of-magnitude savings in proof size and verifier runtime.

Contributions. We design, implement, and evaluate Hyrax, a "doubly" (meaning for both prover and verifier) concretely efficient zkSNARK. For input x, witness w, an AC C of width G and depth d, and a design parameter $\iota \geq 2$ that controls a tradeoff between proof length and verifier time:

- Hyrax's proofs are succinct, i.e., sub-linear in $|C|$ and $|w|$: they require $\approx 10d \log G + |w|^{1/\iota}$ group elements;

- its verifier runs in time sub-linear in $|C|$, if C has sufficient parallelism:[2] $O(|x| + d \log G + |w|^{(\iota-1)/\iota})$, with good constants;

- its prover runs in time linear in $|C|$, with good constants, if C has sufficient parallelism (practically, a few tens of parallel instances suffices), and it requires only $O(d \log G + |w|)$ cryptographic operations, also with good constants; and

- it requires no trusted setup, and it is secure under the discrete log assumption in the random oracle model.

We also give a new commitment scheme tailored to multilinear polynomials (§6), which may be of independent interest. This scheme allows the prover to commit to a multilinear polynomial m over $\mathbb{F}$, and later to reveal (a commitment to) $m(r)$ for any r chosen by the verifier. For $\iota \geq 2$, if $|m|$ denotes the number of monomials in m, then the commitment has size $O(|m|^{1/\iota})$, and the time to verify a purported evaluation is $O(|m|^{(\iota-1)/\iota})$.

2 Related work

ZK proofs. Over the past several years there has been significant interest in implementing ZK proof systems. In this section, we discuss those efforts, focusing on the theoretical underpinnings and associated cryptographic assumptions; we compare Hyrax with several of these works empirically in Section 8.

Gennaro et al. [46] present a linear probabilistically checkable

proof (PCP)[3] and ZK transform that form the basis of many recent zkSNARK implementations [4, 5, 12, 13, 15, 16, 29, 34, 36, 39, 42–44, 64, 76, 79, 105], including systems deployed in applications like ZCash [12, 107]. These implementations build on theoretical work by Ishai et al. [59], Groth [53], Lipmaa [69], and Bitansky et al. [21], as well as implementations and refinements in the non-ZK context [29, 91–93]. Such zkSNARKs give small, constant-sized proofs (hundreds of bytes), and verifier runtime depends only on input size. But ZK systems in this line rely on non-standard, non-falsifiable cryptographic assumptions, require a trusted setup, and have massive prover overhead: runtime is quasi-linear in the verifying circuit size, including a few public key operations per gate, and memory consumption limits the statement sizes these systems can handle in practice [105].

A line of work by Ben-Sasson et al. builds non-interactive ZK arguments from short PCPs, following the seminal work of Kilian [62, 63] and Micali [73], the landmark result of Ben-Sasson and Sudan [17], and recent generalizations of PCPs [9, 14, 87]. The authors reduce the concrete overheads associated with these approaches [10] and implement zero-knowledge scalable transparent arguments of knowledge (zkSTARKs) [11]. zkSTARKs need no trusted setup and no public-key cryptography, but their soundness rests on a non-standard conjecture related to Reed-Solomon codes [11, Appx. B]. Further, zkSTARKs are heavily optimized for statements whose verifying circuits are expressed as a sequence of state-machine transitions; this captures all of NP, but can introduce significant overhead in practice [105]. Both proof size and verifier runtime are logarithmic in circuit size (hundreds of kilobytes and tens of milliseconds, respectively, in practice), and prover runtime is quasi-linear.

Another approach due to Ishai, Kushilevitz, Ostrovsky, and Sahai [60] (IKOS) transforms a secure multi-party computation protocol into a ZK argument. Giacomelli et al. refine this approach and construct ZKBoo [47], a ZK argument system for Boolean circuits with no trusted setup from collision-resistant hashes; ZKB++, by Chase et al. [32], reduces proof size by constant factors. Both schemes are concretely inexpensive for small circuits, but their costs scale linearly with circuit size. Ames et al. [1] further refine the IKOS transform and apply it to a more sophisticated secure computation protocol. Their scheme, Ligero, makes similar security assumptions to ZKBoo but proves an AC C's satisfiability with proof size $\tilde{O}(\sqrt{|C|})$ and prover and verifier work quasi-linear in $|C|$ (where $\tilde{O}$ ignores polylog factors).

Bootle et al. [24] give two ZK arguments for AC satisfiability from the hardness of discrete logarithms, building on the work of Groth [52] and of Bayer and Groth [6]. The first has proof size $O(\sqrt{M})$ and quasi-linear prover and verifier runtime for an AC with M multiplications. The second reduces this to $O(\log M)$ at the cost of concretely longer prover and verifier runtimes. Bünz et al. [30] reduce proof size and runtimes in the log scheme by $\approx 3\times$. Bootle et al. [25] give a ZK argument with proof size $O(\sqrt{|C|})$ whose verifier uses $O(|C|)$ *additions* (which are less expensive than multiplications), but the authors state that the constants are large and do not recommend implementing as-is.

Most similar to our work, Zhang et al. [108] show how to com-

---

[2]Even without parallelism, the verifier runs in time sub-linear in $|C|$ if C's wiring pattern satisfies a technical "regularity" condition [35, 49] (Thm. 1, §3.2).

[3]The observation that the *quadratic span programs* of GGPR [46] can be viewed as linear PCPs is due to Bitansky et al. [21] and Setty et al. [91].

bine an interactive proof [35, 49, 97] and a verifiable polynomial delegation scheme [61, 78] to construct a succinct, non-ZK interactive argument. A follow-up work [109] (concurrent with and independent from ours) achieves ZK using the same commit-and-prove approach that we use, with several key differences. First, their commitment to the witness w has communication $O(\log |w|)$, but has a trusted setup phase and relies on non-standard, non-falsifiable assumptions. In contrast, our commitment protocol (§6) has no trusted setup and is based on the discrete log assumption, but has communication $O(|w|^{1/\iota})$, $\iota \geq 2$. Second, their argument uses an IP that requires more communication than ours (§3.2). Finally, our method of compiling the IP into a ZK argument uses additional refinements (§5) that reduce costs. Both our IP and our refinements apply to their work; we estimate that they would reduce proof size by $\approx 3\times$ and $\mathcal{V}$ runtime by $\approx 5\times$.

Polynomial commitment schemes were introduced by Kate et al. [61], who gave a construction for univariate polynomials based on pairing assumptions. Several follow-up works [78, 108–110] extend this construction to multivariate polynomials; Libert et al. [65] give a construction based on constant-size assumptions; and Fujisaki et al. [45] give a construction for polynomial evaluation based on the RSA problem that can be immediately adapted to polynomial commitment. None of these schemes meet our desiderata (§1) because of some combination of high cost, trusted setup, and non-standard assumptions. Bootle et al. [25] and Bootle and Groth [26] describe univariate polynomial commitment schemes based on the discrete log assumption; our scheme is closely related to these ideas and extends them to multilinear polynomials. The second of these also presents a general framework for proving simple relations between commitments and field elements; exploring these ideas in our context is future work.

3 Background

3.1 Definitions

We use $\langle A(z_a), B(z_b)\rangle(x)$ to denote the random variable representing the (local) output of machine B when interacting with machine A on common input x, when the random tapes for each machine are uniformly and independently chosen, and A and B has auxiliary inputs z_a and z_b respectively. We use $\text{tr}\langle A(z_a), B(z_b)\rangle(x)$ to denote the random variable representing the entire transcript of the interaction between A and B, and $\text{View}(\langle A(z_a), B(z_b)\rangle(x))$ to denote the distribution of the transcript. The symbol $\approx_c$ denotes that two ensembles are computationally indistinguishable.

Arithmetic circuits

Section 3.2 considers the arithmetic circuit (AC) *evaluation* problem. In this problem, one fixes an arithmetic circuit C, consisting of addition and multiplication gates over a finite field $\mathbb{F}$. We assume throughout that C is layered, with all gates having fan-in at most 2 (any arithmetic circuit can be made layered while increasing the number of gates by a factor of at most the circuit depth). C has depth d and input x with length $|x|$. The goal is to evaluate C on input x. In an interactive proof or argument for this problem, the prover sends the claimed outputs y of C on input x, and must prove that $y = C(x)$.

Our end goal in this work is to give efficient protocols for the arithmetic circuit *satisfiability* problem. Let $C(\cdot, \cdot)$ be a layered arithmetic circuit of fan-in two. Given an input x and outputs y, the goal is to determine whether there exists a *witness* w such that $C(x, w) = y$. The corresponding witness relation for this problem is the natural one: $R_{(x,y)} = \{w : C(x, w) = y\}$.

Interactive protocols and zero-knowledge

Definition 1 (Interactive arguments and proofs). *A pair of probabilistic interactive machines $\langle \mathcal{P}, \mathcal{V}\rangle$ is called an interactive argument system for a language L if there exists a negligible function η such that the following two conditions hold:*

1. *Completeness: For every $x \in L$ there exists a string w s.t. for every $z \in \{0, 1\}^*$, $\Pr[\langle \mathcal{P}(w), \mathcal{V}(z)\rangle(x)=1] \geq 1 - \eta(|x|)$.*

2. *Soundness: For every $x \notin L$, every interactive PPT $\mathcal{P}^*$, and every $w, z \in \{0, 1\}^*$, $\Pr[\langle \mathcal{P}^*(w), \mathcal{V}(z)\rangle(x)=1] \leq \eta(|x|)$.*

If soundness holds against computationally unbounded cheating provers $\mathcal{P}^*$, then $\langle \mathcal{P}, \mathcal{V}\rangle$ is called an interactive proof (IP).

Definition 2 (Zero-knowledge (ZK)). *Let $L \subset \{0, 1\}^*$ be a language and for each $x \in L$, let $R_x \subset \{0, 1\}^*$ denote a corresponding set of* witnesses *for the fact that $x \in L$. Let R_L denote the corresponding language of valid (input, witness) pairs, i.e., $R_L = \{(x, w) : x \in L \text{ and } w \in R_x\}$. An interactive proof or argument system $\langle \mathcal{P}, \mathcal{V}\rangle$ for L is computational zero-knowledge (CZK) with respect to an auxiliary input if for every PPT interactive machine $\mathcal{V}^*$, there exists a PPT algorithm S, called the* simulator, *running in time polynomial in the length of its first input, such that for every $x \in L$, $w \in R_x$, and $z \in \{0, 1\}^*$,*

$$\text{View}(\langle P(w), \mathcal{V}^*(z)\rangle(x)) \approx_c S(x, z) \qquad (1)$$

when the distinguishing gap is considered as a function of $|x|$. If the statistical distance *between the two distributions is negligible, then the interactive proof or argument system is said to be* statistical *zero-knowledge (SZK). If the simulator is allowed to abort with probability at most $1/2$, but the distribution of its output conditioned on not aborting is* identically *distributed to* $\text{View}(\langle P(w), \mathcal{V}^*(z)\rangle(x))$, *then the interactive proof or argument system is called* perfect *zero-knowledge (PZK).*

The left term in Equation (1) denotes the distribution of transcripts after $\mathcal{V}^*$ interacts with $\mathcal{P}$ on common input x; the right term denotes the distribution of simulator S's output on x. For any CZK (resp., SZK or PZK) protocol, Definition 2 requires the simulator to produce a distribution that is computationally (resp., statistically or perfectly) indistinguishable from the distribution of transcripts of the ZK proof or argument system.

Our zero-knowledge arguments also satisfy a proof of knowledge property. Intuitively, this means that in order to produce a convincing proof of a statement, the prover must *know* a witness to the validity of the statement. To define this notion formally, we follow Groth and Ishai [54] who borrow the notion of statistical witness-extended emulation from Lindell [68]:

Definition 3 (Witness-extended emulation [54]). *Let L be a language and R_L corresponding language of valid (input, witness) pairs as in Definition 2. An interactive argument system $\langle \mathcal{P}, \mathcal{V}\rangle$ for L has* witness-extended emulation *if for all deterministic polynomial time $\mathcal{P}^*$ there exists an expected polynomial time emulator E such that for all non-uniform polynomial time adversaries A and all $z_\mathcal{V} \in \{0, 1\}^*$, the following probabilities differ*

by at most a negligible function in the security parameter λ:

$$\Pr\left[(x, z_{\mathcal{P}}) \leftarrow A(1^\lambda); t \leftarrow \mathrm{tr}\langle \mathcal{P}^*(z_{\mathcal{P}}), \mathcal{V}(z_{\mathcal{V}})\rangle(x) : A(t) = 1\right]$$

$$and \ \Pr\left[\begin{array}{l} (x, z_{\mathcal{P}}) \leftarrow A(1^\lambda); (t, w) \leftarrow E^{\mathcal{P}^*(z_{\mathcal{P}})}(x) : A(t) = 1 \ \wedge \\ \textit{if } t \textit{ is an accepting transcript, then } (x, w) \in R_L. \end{array}\right]$$

Here, the oracle called by E permits rewinding the prover to a specific point and resuming with fresh randomness for the verifier from this point onwards.

Commitment schemes

Informally, a commitment scheme allows a *sender* to produce a message $C = \mathsf{Com}(m)$ that hides m from a *receiver* but binds the sender to the value m. In particular, when the sender *opens C* and reveals m, the receiver is convinced that this was indeed the sender's original value. We say that $\mathsf{Com}_{\mathrm{pp}}(m; r)$ is a commitment to m with *opening r* with respect to public parameters pp. The sender chooses r at random; to open the commitment, the sender reveals (m, r). We frequently leave the public parameters implicit, and sometimes do the same for the opening, e.g., $\mathsf{Com}(m)$.

Definition 4 (Collection of non-interactive commitments [57])**.** *We say that a tuple of PPT algorithms* (Gen, Com) *is a* collection of non-interactive commitments *if the following conditions hold:*

- **Computational binding:** *For every (non-uniform) PPT A, there is a negligible function η such that for every $n \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{l} \mathrm{pp} \leftarrow \mathsf{Gen}(1^n) \ ; \\ (m_0, r_0), (m_1, r_1) \leftarrow A(1^n, \mathrm{pp}) \ : \\ m_0 \neq m_1, |m_0| = |m_1| = n, \\ \mathsf{Com}_{\mathrm{pp}}(m_0; r_0) = \mathsf{Com}_{\mathrm{pp}}(m_1; r_1) \end{array}\right] \leq \eta(n)$$

- **Perfect hiding:** *For any* pp $\in \{0, 1\}^*$ *and $m_0, m_1 \in \{0, 1\}^*$ where $|m_0| = |m_1|$, the ensembles $\{\mathsf{Com}_{\mathrm{pp}}(m_0)\}_{n \in \mathbb{N}}$ and $\{\mathsf{Com}_{\mathrm{pp}}(m_1)\}_{n \in \mathbb{N}}$ are identically distributed.*

Collections of non-interactive commitments can be constructed based on any one-way function [56, 75], but we require a *homomorphism* property (defined below) that these commitments do not provide. (The Pedersen commitment [80], described in Appx. A, provides this property.)

Definition 5 (Additive homomorphism)**.** *Given* $\mathsf{Com}(x; s_x)$ *and* $\mathsf{Com}(y; s_y)$, *there is an operator $\odot$ such that*

$$\mathsf{Com}(x; s_x) \odot \mathsf{Com}(y; s_y) = \mathsf{Com}(x + y; s_x + s_y) \qquad and$$

$$\mathsf{Com}(x; s_x)^k \triangleq \mathsf{Com}(x; s_x) \odot \cdots \odot \mathsf{Com}(x; s_x) \qquad (k \ times)$$

In a multi-commitment *scheme, x and y are vectors, and this additive homomorphism is vector-wise.*

3.2 Our starting point: Gir^{++} (Giraffe, with a tweak)

The most efficient known IPs for the AC evaluation problem (§3.1) follow a line of work starting with the breakthrough result of Goldwasser, Kalai, and Rothblum (GKR) [49]. Cormode, Mitzenmacher, and Thaler (CMT) [35] and Vu et al. [102] refine this result, giving $O(|C| \log |C|)$ prover and $O(|x| + |y| + d \log |C|)$ verifier runtimes, for AC C with depth d, input x, and output y.

Further refinements are possible in the case where C is *data parallel*, meaning it consists of N identical sub-computations run on different inputs. (We refer to each sub-computation as a

sub-AC of C, and we assume for simplicity that all layers of the sub-AC have width G, so $|C| = d \cdot N \cdot G$.) Thaler [97] reduced the prover's runtime in the data-parallel case from $O(|C| \log |C|)$ to $O(|C| \log G)$. Very recently, Wahby et al. introduced Giraffe [104], which reduces the prover's runtime to $O(|C| + d \cdot G \cdot \log G)$. Since $|C| = d \cdot N \cdot G$, observe that when $N \geq \log G$, the time reduces to $O(|C||)$, which is asymptotically optimal. That is, for sufficient data parallelism, the prover's runtime is just a constant factor slower than evaluating the circuit gate-by-gate without providing any proof of correctness.

Our work builds on Gir^{++}, which reduces Giraffe's communication via an optimization due to Chiesa et al. [33]; our description of Gir^{++} borrows notation from Wahby et al. [104]. Assume for simplicity that N and G are powers of 2, and let $b_N = \log_2 N$ and $b_G = \log_2 G$. Within a layer of C, each gate is labeled with a pair $(i, j) \in \{0, 1\}^{b_N} \times \{0, 1\}^{b_G}$. Number the layers of C from 0 to d in reverse execution order, so that 0 refers to the output layer, and d refers to the input layer. Each layer i is associated with an evaluator function $V_i \colon \{0, 1\}^{b_N} \times \{0, 1\}^{b_G} \to \mathbb{F}$ that maps a gate's label to the output of that gate when C is evaluated on input x. For example, $V_0(i, j)$ is the j'th output of the i'th sub-AC, and $V_d(i, j)$ is the jth input to the ith sub-AC.

At a high level, the protocol proceeds in iterations, one for each layer of the circuit. At the start of the protocol, the prover $\mathcal{P}$ sends the claimed outputs y of C (i.e., all the claimed evaluations of V_0). The first iteration of the protocol reduces the claim about V_0 to a claim about V_1, in the sense that it is safe for the verifier $\mathcal{V}$ to believe the former claim as long as $\mathcal{V}$ is convinced of the latter. But $\mathcal{V}$ cannot directly check the claim about V_1, because doing so would require evaluating all of the gates in C other than the outputs themselves. Instead, the second iteration reduces the claim about V_1 to a claim about V_2, and so on, until $\mathcal{P}$ makes a claim about V_d (i.e., the inputs to C), which $\mathcal{V}$ checks itself.

To describe how the reduction from a claim about V_i to a claim about V_{i+1} is performed, we first introduce *multilinear extensions*, *the sum-check protocol*, and *wiring predicates*.

Multilinear extensions. An *extension* of a function $f \colon \{0, 1\}^\ell \to \mathbb{F}$ is a ℓ-variate polynomial g over $\mathbb{F}$ such that $g(x) = f(x)$ for all $x \in \{0, 1\}^\ell$. Any such function f has a unique multilinear extension (MLE)—a multilinear polynomial—denoted $\tilde{f}$. Given a vector $z \in \mathbb{F}^m$ with $m = 2^\ell$, we will often view z as a function $z \colon \{0, 1\}^\ell \to \mathbb{F}$ mapping indices to vector entries, and use $\tilde{z}$ to denote the MLE of z.

The sum-check protocol. Fix an ℓ-variate polynomial g over $\mathbb{F}$, and let $\deg_i(g)$ denote the degree of g in variable i. The sum-check protocol [70] is an interactive proof that allows $\mathcal{P}$ to convince $\mathcal{V}$ of a claim about the value of $\sum_{x \in \{0,1\}^\ell} g(x)$ by reducing it to a claim about the value of $g(r)$, where $r \in \mathbb{F}^\ell$ is a point randomly chosen by $\mathcal{V}$. There are ℓ rounds, and $\mathcal{V}$'s runtime is $O(\sum_{i=1}^\ell \deg_i(g))$ plus the cost of evaluating $g(r)$. The mechanics are detailed in Section 4.

Wiring predicates capture the wiring information of the sub-ACs. Define the wiring predicate $\mathsf{add}_i \colon \{0, 1\}^{3b_G} \to \{0, 1\}$, where $\mathsf{add}_i(g, h_0, h_1)$ returns 1 if (a) within each sub-AC, gate g at layer $i - 1$ is an add gate and (b) the left and right inputs of g are, respectively, h_0 and h_1 at layer i (and 0 otherwise). mult_i is

defined analogously for multiplication gates. Define the equality predicate $\mathrm{eq}: \{0,1\}^{2b_N} \to \{0,1\}$ as $\mathrm{eq}(a,b) = 1$ iff $a = b$.

Thaler [97, 98] and Wahby et al. [104] show how to express $\tilde{V}_{i-1}$ in terms of $\tilde{V}_i$: for $(q',q) \in \mathbb{F}^{b_N} \times \mathbb{F}^{b_G}$, let $P_{q',q,i}: \mathbb{F}^{b_N} \times \mathbb{F}^{b_G} \times \mathbb{F}^{b_G} \to \mathbb{F}$ denote the polynomial

$$P_{q',q,i}(h', h_L, h_R) =$$
$$\widetilde{\mathrm{eq}}(q',h') \cdot \big[\widetilde{\mathrm{add}}_i(q, h_L, h_R) \left(\tilde{V}_i(h', h_L) + \tilde{V}_i(h', h_R) \right) +$$
$$\widetilde{\mathrm{mult}}_i(q, h_L, h_R) \left(\tilde{V}_i(h', h_L) \cdot \tilde{V}_i(h', h_R) \right) \big]$$

Then we have

$$\tilde{V}_{i-1}(q',q) = \sum_{h' \in \{0,1\}^{b_N}} \sum_{h_L, h_R \in \{0,1\}^{b_G}} P_{q',q,i}(h', h_L, h_R). \quad (2)$$

Protocol overview

Step 1. At the start of the protocol, $\mathcal{P}$ sends the claimed output y, thereby specifying a function $V_y: \{0,1\}^{b_G+b_N} \to \mathbb{F}$ mapping the label of each output gate to the corresponding entry of y. The verifier wishes to check that $V_y = V_0$ (i.e., that the claimed outputs equal the correct outputs of C on input x); to accomplish this, it would be enough to check that $\tilde{V}_y = \tilde{V}_0$. In principle, $\mathcal{V}$ could do that by choosing a random pair $(q',q) \in \mathbb{F}^{b_N} \times \mathbb{F}^{b_G}$ and checking that $\tilde{V}_y(q',q) = \tilde{V}_0(q',q)$; if that check passes, then $\tilde{V}_y = \tilde{V}_0$ with high probability, by the Schwartz-Zippel lemma. On the one hand, $\mathcal{V}$ can and does compute $\tilde{V}_y(q',q)$; this takes $O(NG)$ time [104, §3.3]. But on the other hand, $\mathcal{V}$ cannot compute $\tilde{V}_0(q',q)$ directly—this would require $\mathcal{V}$ to evaluate C.

Step 2 (iterated). Instead, $\mathcal{V}$ outsources evaluation of $\tilde{V}_0(q',q)$ to $\mathcal{P}$, via the sum-check protocol; this is motivated by Equation (2). At the end of the sum-check protocol, $\mathcal{V}$ must evaluate $P_{q',q,1}$ at a random input (r', r_L, r_R), which requires the values $\tilde{V}_1(r', r_L)$ and $\tilde{V}_1(r', r_R)$. $\mathcal{V}$ does not evaluate these points directly; that would be too costly. Instead, $\mathcal{P}$ sends v_0 and v_1, which it claims are the required values. $\mathcal{V}$ uses these to evaluate $P_{q',q,1}$, then checks v_0 and v_1 using a *mini-protocol*, which we describe shortly. At a high level, the mini-protocol transforms $\mathcal{P}$'s claims about v_0, v_1 into a claim about $\tilde{V}_2$. $\mathcal{V}$ checks this claim with a sum-check and mini-protocol invocation, yielding a claim about $\tilde{V}_3$. $\mathcal{P}$ and $\mathcal{V}$ iterate, layer by layer, until $\mathcal{V}$ has a claim about $\tilde{V}_d$.

Final step. $\mathcal{V}$ checks $\mathcal{P}$'s final claim about $\tilde{V}_d$ by evaluating $\tilde{V}_x$ (since $\tilde{V}_d = \tilde{V}_x$); it can do this in $O(NG)$ time [104, §3.3].

Mini-protocols: reducing from $\tilde{V}_i$ to $\tilde{V}_{i+1}$

Gir^{++} differs from Giraffe only in that they use different mini-protocols to reduce $\mathcal{P}$'s claims at the end of one sum-check invocation (i.e., $v_0 = \tilde{V}_i(r', r_L)$ and $v_1 = \tilde{V}_i(r', r_R)$) into the expression that $\mathcal{V}$ and $\mathcal{P}$ use for the next sum-check invocation.

Reducing from two points to one point. This approach is used in Giraffe and prior work [35, 49, 97, 102–104]. $\mathcal{P}$ sends $\mathcal{V}$ the restriction of $\tilde{V}_i$ to the unique line H in $\mathbb{F}^{b_N+b_G}$ passing through the points (r', r_L) and (r', r_R) by specifying the univariate polynomial $f_H(t) = \tilde{V}_i(r', (1-t) \cdot r_L + t \cdot r_R)$, which has degree b_G. $\mathcal{V}$ should believe this claim as long as $f_H(0) = v_0$, $f_H(1) = v_1$, and $f_H(\upsilon) = \tilde{V}_i(r', r_\upsilon)$, where $r_\upsilon = (1-\upsilon) \cdot r_L + \upsilon \cdot r_R$ and υ is chosen by $\mathcal{V}$. By Equation (2), $\mathcal{V}$ can check this latter equality by engaging $\mathcal{P}$ in a sum-check protocol over $P_{r',r_\upsilon,i+1}$.

Alternative: Random linear combination. Each invocation of the prior mini-protocol requires $\mathcal{P}$ to send $b_G + 1$ field elements specifying f_H. The following technique, due to Chiesa et al. [33], eliminates this requirement. Instead, $\mathcal{V}$ checks v_0 and v_1 by checking a random linear combination, via a sum-check invocation over a polynomial we define below.

In more detail, $\mathcal{V}$ samples two field elements μ_0 and μ_1, and sends them to $\mathcal{P}$. Mechanically, $\mathcal{V}$ next checks that

$$\mu_0 \cdot \tilde{V}_i(r', r_L) + \mu_1 \cdot \tilde{V}_i(r', r_R) = \mu_0 \cdot v_0 + \mu_1 \cdot v_1 \quad (3)$$

since, by the Schwartz-Zippel lemma, this implies that $v_0 = \tilde{V}_i(r', r_L)$ and $v_1 = \tilde{V}_i(r', r_R)$ with high probability (formalized in Thm. 1, below). $\mathcal{V}$ checks Equation (3) by exploiting the fact that its LHS can be written as

$$\mu_0 \cdot \tilde{V}_i(q', q_L) + \mu_1 \cdot \tilde{V}_i(q', q_R)$$
$$= \sum_{h_L, h_R \in \{0,1\}^{b_G}} \sum_{h' \in \{0,1\}^{b_N}} \big[\mu_0 \cdot P_{q', q_L, i+1}(h', h_L, h_R) +$$
$$\mu_1 \cdot P_{q', q_R, i+1}(h', h_L, h_R) \big]$$
$$= \sum_{h_L, h_R \in \{0,1\}^{b_G}} \sum_{h' \in \{0,1\}^{b_N}} Q_{q', q_L, q_R, \mu_0, \mu_1, i+1}(h', h_L, h_R)$$

where $Q_{q', q_L, q_R, \mu_0, \mu_1, i}: \mathbb{F}^{b_N} \times \mathbb{F}^{b_G} \times \mathbb{F}^{b_G} \to \mathbb{F}$ is given by:

$$Q_{q', q_L, q_R, \mu_0, \mu_1, i}(h', h_L, h_R) \triangleq \widetilde{\mathrm{eq}}(q', h') \cdot$$
$$\big[\left(\mu_0 \cdot \widetilde{\mathrm{add}}_i(q_L, h_L, h_R) + \mu_1 \cdot \widetilde{\mathrm{add}}_i(q_R, h_L, h_R) \right) \cdot$$
$$\left(\tilde{V}_i(h', h_L) + \tilde{V}_i(h', h_R) \right)$$
$$+ \left(\mu_0 \cdot \widetilde{\mathrm{mult}}_i(q_L, h_L, h_R) + \mu_1 \cdot \widetilde{\mathrm{mult}}_i(q_R, h_L, h_R) \right) \cdot$$
$$\left(\tilde{V}_i(h', h_L) \cdot \tilde{V}_i(h', h_R) \right) \big]$$

This means that $\mathcal{V}$ can check that Equation (3) holds by engaging $\mathcal{P}$ in a sum-check protocol over $Q_{r', r_L, r_R, \mu_0, \mu_1, i+1}$.

Giraffe vs. Gir++ Gir^{++} uses the Alternative above. This reduces communication cost in Gir^{++} compared to Giraffe by a small factor that depends on the amount of data parallelism. We are motivated to reduce communication because communication will translate into proof size and more cryptographic cost (§4).

As an exception, Gir^{++} uses the "reducing from two points to one point" technique after the final sum-check (i.e., the one over $Q_{\dots,d-1}$); this is to avoid increasing $\mathcal{V}$'s computational costs compared to Giraffe. Recall that in the final step of Gir^{++}, $\mathcal{V}$ checks $\mathcal{P}$'s claim about $\tilde{V}_d$ by evaluating $\tilde{V}_x$ (which is equal to $\tilde{V}_d$). Thus, to check the LHS of Equation (3), $\mathcal{V}$ would require *two* evaluations of $\tilde{V}_x$; the "reducing from two points to one point" technique requires only one. Since evaluating $\tilde{V}_x$ is typically a bottleneck for the verifier [104, §3.3], eliminating the second evaluation is worthwhile even though it slightly increases the size of $\mathcal{P}$'s final message (and thus the proof size; see §4).

We give pseudocode for Gir^{++} in the full version [106, Appx. E]. Gir^{++}'s efficiency and security are formalized in the following theorem, which can be proved via a standard analysis [49].

Theorem 1. *The interactive proof Gir^{++} satisfies the following properties when applied to a layered arithmetic circuit C of fan-in two, consisting of N identical sub-computations, each of depth d, with all layers of each sub-computation having*

width at most G. It has perfect completeness, and soundness error at most $((1 + 2 \log G + 3 \log N) \cdot d + \log G)/|\mathbb{F}|$. After a pre-processing phase taking time $O(dG)$, the verifier runs in time $O(|x| + |y| + d \log NG)$, and the prover runs in time $O(|C| + d \cdot G \cdot \log G)$. If the sub-AC has a regular wiring pattern as defined in [35], then the pre-processing phase is unnecessary.

4 Compiling Gir^{++} into a ZK argument

In this section, we describe a straightforward application of "commit-and-prove" techniques [8, 37] (§1) to Gir^{++} (§3.2). The result is a public coin, perfect ZK argument "of knowledge" for AC satisfiability (the knowledge property is formalized via witness-extended emulation; §3.1). In Sections 5 and 6, we develop substantial efficiency improvements; in Section 7, we apply the Fiat-Shamir heuristic [41] to make it non-interactive.

Building blocks. This section uses abstract commitments having a homomorphism property (§3.1). We also make black-box use of three sub-protocols, which operate on commitments:

- *proof-of-opening*(C) convinces $\mathcal{V}$ that $\mathcal{P}$ can open C.

- *proof-of-equality*(C_0, C_1) convinces $\mathcal{V}$ that C_0 and C_1 commit to the same value, and that $\mathcal{P}$ can open both.

- *proof-of-product*(C_0, C_1, C_2) convinces $\mathcal{V}$ that C_2 commits to the product of the values committed in C_0 and C_1, and that $\mathcal{P}$ can open all three.

In Appendix A, we give concrete definitions of the above protocols in terms of Pedersen commitments [80].

Protocol overview. This protocol differs from Gir^{++} in three ways. First, it adds an initial step in which $\mathcal{P}$ commits to w such that $C(x, w) = y$. Second, $\mathcal{P}$ replaces all of its messages in Gir^{++} with *commitments* to those messages. Third, $\mathcal{P}$ convinces $\mathcal{V}$ that its committed values pass all of $\mathcal{V}$'s checks in Gir^{++} using the homomorphism property of the commitments and the above sub-protocols. The steps below correspond to the steps of Gir^{++} (§3.2); we describe only how the protocols differ.

Step 0. (This is a new step.) $\mathcal{P}$ sends commitments to each element of $w \in \mathbb{F}^\ell$. $\mathcal{P}$ and $\mathcal{V}$ execute proof-of-opening for each.

Step 1. As in Gir^{++}, $\mathcal{V}$ computes $\tilde{V}_y(q', q)$. Afterwards, $\mathcal{V}$ computes $C_0 = \mathsf{Com}(\tilde{V}_y(q', q); 0)$.

Step 2. As in Gir^{++}, this step comprises one sum-check and one mini-protocol per layer of C. We now review the sum-check protocol, and then describe how $\mathcal{P}$ and $\mathcal{V}$ execute the sum-check and mini-protocols "underneath the commitments."

Review of the sum-check protocol. We begin by describing the first layer sum-check protocol in Gir^{++} (others are similar), which reduces $\tilde{V}_y(q', q)$ to a claim about $\tilde{V}_1(\cdot)$. In the first round of the sum-check protocol, $\mathcal{P}$ sends a univariate polynomial $s_1(\cdot)$ of degree 3. $\mathcal{V}$ checks that $s_1(0) + s_1(1) = \tilde{V}_y(q', q)$, and then sends a random field element r_1 to $\mathcal{P}$. In general, in round j of the sum-check protocol, $\mathcal{P}$ sends a univariate polynomial s_j (which is degree 3 in the first b_N rounds and degree 2 in the remaining rounds [97, 104]). $\mathcal{V}$ checks that $s_j(0) + s_j(1) = s_{j-1}(r_{j-1})$, then sends a random field element r_j to $\mathcal{P}$.

We write the vector of all r_j's chosen by $\mathcal{V}$ in the $j_{\text{last}} = b_N + 2b_G$ rounds of the sum-check protocol as $(r_1, \ldots, r_{j_{\text{last}}}) \in$

$\mathbb{F}^{b_N + 2b_G}$; let r' denote the first b_N entries of this vector, r_L denote the next b_G entries, and r_R denote the final b_G entries.

In the last round, $\mathcal{P}$ sends v_0 and v_1 (which it claims are equal to $\tilde{V}_1(r', r_L)$ and $\tilde{V}_1(r', r_R)$; §3.2). $\mathcal{V}$ first checks that

$$s_{j_{\text{last}}}(r_{j_{\text{last}}}) = \tilde{\mathsf{eq}}(q', r') \cdot \big[\tilde{\mathsf{add}}_1(q, r_L, r_R) \cdot (v_0 + v_1) + \\ \tilde{\mathsf{mult}}_1(q, r_L, r_R) \cdot v_0 \cdot v_1 \big]$$

$\mathcal{V}$ then checks $\mathcal{P}$'s claims about v_0 and v_1 by invoking a mini-protocol (§3.2) and engaging $\mathcal{P}$ in another sum-check at layer 2.

ZK sum-check protocol. In round j of the sum-check, $\mathcal{P}$ commits to $s_j(t) = c_{0,j} + c_{1,j}t + c_{2,j}t^2 + c_{3,j}t^3$, via $\delta_{c_{0,j}} \leftarrow \mathsf{Com}(c_{0,j})$, $\delta_{c_{1,j}} \leftarrow \mathsf{Com}(c_{1,j})$, $\delta_{c_{2,j}} \leftarrow \mathsf{Com}(c_{2,j})$, and $\delta_{c_{3,j}} \leftarrow \mathsf{Com}(c_{3,j})$, and $\mathcal{P}$ and $\mathcal{V}$ execute proof-of-opening for each one. Now $\mathcal{P}$ convinces $\mathcal{V}$ that $s_j(0) + s_j(1) = s_{j-1}(r_{j-1})$. Notice that if $\mathcal{V}$ holds commitments $\mathsf{Com}(s_{j-1}(r_{j-1}))$ and $\mathsf{Com}(s_j(0) + s_j(1))$, $\mathcal{P}$ can use proof-of-equality to convince $\mathcal{V}$ that the above equation holds. Further, $\mathcal{V}$ can use the homomorphism property to compute the required commitments: for $s_j(0) + s_j(1) = 2c_{0,j} + c_{1,j} + c_{2,j} + c_{3,j}$, $\mathcal{V}$ computes $\delta_{c_{0,j}}^2 \odot \delta_{c_{1,j}} \odot \delta_{c_{2,j}} \odot \delta_{c_{3,j}}$. Similarly, for $s_{j-1}(r_{j-1})$ $\mathcal{V}$ computes $\delta_{c_{0,j}} \odot \delta_{c_{1,j}}^{r_j} \odot \delta_{c_{2,j}}^{r_j^2} \odot \delta_{c_{3,j}}^{r_j^3}$.

The first sum-check round ($j = 1$) is an exception to the above: rather than a commitment to s_0, $\mathcal{V}$ holds a commitment to a value that purportedly equals $s_1(0) + s_1(1)$. For the sum-check invocation at layer 1, this value is C_0, which $\mathcal{V}$ computed in Step 1. For subsequent layers, the value is the result of the preceding mini-protocol invocation, which we discuss below.

In the final round j_{last}, $\mathcal{V}$ computes a commitment W to $s_{j_{last}}(r_{j_{\text{last}}})$ as described above. $\mathcal{P}$ then sends commitments X, Y, and Z to v_0, v_1, and $v_0 \cdot v_1$, and uses proof-of-product to convince $\mathcal{V}$ that the committed values satisfy this product relation. Finally, $\mathcal{V}$ computes $\Omega \leftarrow (X \odot Y)^{\tilde{\mathsf{eq}}(q',r') \cdot \tilde{\mathsf{add}}_1(q,r_1,r_2)} \odot Z^{\tilde{\mathsf{eq}}(q',r') \cdot \tilde{\mathsf{mult}}_1(q,r_1,r_2)}$ and $\mathcal{P}$ uses proof-of-equality to convince $\mathcal{V}$ that W and Ω commit to the same value.

ZK mini-protocols. For random-linear-combination, $\mathcal{V}$ computes $\mathsf{Com}(\mu_0 v_0 + \mu_1 v_1) = X^{\mu_0} \odot Y^{\mu_1}$; this is the purported $\mathsf{Com}(s_1(0) + s_1(1))$ for the next sum-check invocation.

To execute reducing-from-two-points-to-one-point, $\mathcal{P}$ commits to the coefficients of f_H and invokes proof-of-opening for each; $\mathcal{V}$ computes commitments to $f_H(0)$ and $f_H(1)$, and $\mathcal{P}$ uses proof-of-equality to show that these commit to the same values as X and Y; and $\mathcal{V}$ samples υ and computes a commitment to $f_H(\upsilon)$, which it uses in the final step.

Final step. $\mathcal{P}$ now convinces $\mathcal{V}$ that $\mathsf{Com}(f_H(\upsilon))$, the result of the final mini-protocol invocation (which is a commitment to $\tilde{V}_d(r', r_\upsilon)$; §3.2), is consistent with x and w.

We let $m = (x, w)$ denote the concatenation of the input x and the witness w; assume for simplicity that $|x| = |w| = 2^\ell$; interpret x, w, and m as functions (§3.2, "Multilinear extensions"); and let $(r_0, \ldots, r_\ell) = (r', r_\upsilon)$. Then by the definitions of $\tilde{m}$, $\tilde{x}$, and $\tilde{w}$,

$$\tilde{m}(r_0, \ldots, r_\ell) = (1 - r_0) \cdot \tilde{x}(r_1, \ldots, r_\ell) + r_0 \cdot \tilde{w}(r_1, \ldots, r_\ell).$$

By analogy to Gir^{++}'s final step, $\mathcal{V}$'s task is to check that $\tilde{V}_d(r', r_\upsilon)$ is equal to $\tilde{m}(r_0, \ldots, r_\ell)$. $\mathcal{V}$ does this by first computing $\mathsf{Com}(\tilde{m}(r_0, \ldots, r_\ell))$ using the commitments to w that $\mathcal{P}$ sent in Step 0 (above), and then engaging $\mathcal{P}$ in proof-of-equality on $\mathsf{Com}(f_H(\upsilon))$ and $\mathsf{Com}(\tilde{m}(r_0, \ldots, r_\ell))$).

To compute $\mathsf{Com}(\tilde{m}(\cdot))$, $\mathcal{V}$ exploits the following expression [35] for the multilinear extension of $w \colon \{0,1\}^\ell \to \mathbb{F}$:

$$\tilde{w}(r_1, \ldots, r_\ell) = \sum_{b \in \{0,1\}^\ell} w(b) \cdot \prod_{k \in \{1, \ldots, \ell\}} \chi_{b_k}(r_k)$$
$$= \sum_{b \in \{0,1\}^\ell} w(b) \cdot \chi_b \qquad (4)$$

where $\chi_{b_k}(r_k) = r_k b_k + (1 - r_k)(1 - b_k)$, $\chi_b = \prod_k \chi_{b_k}(r_k)$, and b_k is the (1-indexed) k^{th} bit of b. In more detail, $\mathcal{V}$ first evaluates each χ_b in linear time [102], and then computes

$$F = \bigodot_{b \in \{0,1\}^\ell} \mathsf{Com}(w(b))^{r_0 \cdot \chi_b(r_1, \ldots, r_\ell)}$$

which is $\mathsf{Com}(r_0 \cdot \tilde{w}(r_1, \ldots, r_\ell))$. It then computes, in the clear, $F' = (1 - r_0) \cdot \tilde{x}(r_1, \ldots, r_\ell)$. Finally, $\mathcal{V}$ computes $\mathsf{Com}(\tilde{m}(r_0, \ldots, r_\ell)) = F \odot \mathsf{Com}(F'; 0)$. Invoking proof-of-equality as described above completes the protocol.

The following theorem formalizes the efficiency of the argument of this section. We leave a formal statement of security properties to the final protocol (§7).

Theorem 2. *Let $C(\cdot, \cdot)$ be a layered arithmetic circuit of fan-in two, consisting of N identical sub-computations, each of depth d, with all layers of each sub-computation having width at most G. Assuming the existence of computationally binding, perfectly hiding homomorphic commitment schemes that support proof-of-opening, proof-of-equality, and proof-of-product (Appx. A) with running times upper-bounded by κ, there exists a PZK argument for the NP relation "$\exists w$ such that $C(x, w) = y$." The protocol requires $d \log(G)$ rounds of communication, and has communication complexity $\Theta(|y| + (|w| + d \log G) \cdot \lambda)$, where λ is a security parameter. Given a w such that $C(x, w) = y$, the prover runs in time $\Theta(dNG + G \log G + (|w| + d \log G) \cdot \kappa)$. Verifier runtime is $\Theta(|x| + |y| + dG + (|w| + d \log(NG)) \cdot \kappa)$.*

The above follows from the more general Theorem 3.1 of [8].

5 Reducing the cost of sum-checks

In the PZK argument from Section 4, the prover sends a separate commitment for every message element of Gir^{++} (§3.2), and then independently proves knowledge of how to open each commitment. This leads to long proofs and many expensive cryptographic operations for the verifier.

In this section, we explain how to reduce this communication and the number of cryptographic operations for the verifier by exploiting *multi-commitment* schemes, in which a commitment to a vector of elements has the same size as a commitment to a single element. The Pedersen commitment [80] (Appx. A) supports multi-commitments.

Dot-product proof protocol. Our starting point is an existing protocol for multi-commitments, which we call proof-of-dot-prod. With this protocol, a prover that knows the openings of two commitments, one to a vector $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{F}^n$ and one to a scalar $y \in \mathbb{F}$, can prove in zero-knowledge that $y = \langle \vec{a}, \vec{x} \rangle$ for a public $\vec{a} \in \mathbb{F}^n$. The protocol is defined in Appendix A.2.

Squashing $\mathcal{V}$'s checks. To exploit proof-of-dot-prod, we first recall from Section 4 that in each round j of each sum-check invocation in Gir^{++}, $\mathcal{P}$ sends commitments to $c_{0,j}$,

$c_{1,j}$, $c_{2,j}$, and (only in the first b_N rounds) $c_{3,j}$. Next, $\mathcal{P}$ proves to $\mathcal{V}$ that $2c_{0,j} + c_{1,j} + c_{2,j} + c_{3,j} = s_{j-1}(r_{j-1})$ (i.e., that $s_j(0) + s_j(1) = s_{j-1}(r_{j-1})$). Finally, $\mathcal{V}$ computes a commitment to $s_j(r_j) = c_{0,j} + c_{1,j}r_j + c_{2,j}r_j^2 + c_{3,j}r_j^3$ for the next round.

Combining the above equations yields $c_{3,j+1} + c_{2,j+1} + c_{1,j+1} + 2c_{0,j+1} - (c_{3,j}r_j^3 + c_{2,j}r_j^2 + c_{1,j}r_j + c_{0,j}) = 0$. $\mathcal{V}$'s final check can likewise be expressed as a linear equation in terms of $v_0, v_1, c_{2,n}, c_{1,n}, c_{0,n}$, and wiring predicate evaluations (§3.2) ($n = b_N + 2b_G$). We can thus write $\mathcal{V}$'s checks during the rounds of the sum-check protocol as the matrix-vector product

$$\begin{bmatrix} M_1 \\ \vdots \\ M_{b_N+2b_G+1} \end{bmatrix} \cdot \vec{\pi} = \begin{bmatrix} s_0 \\ 0 \\ \vdots \end{bmatrix} \qquad (5)$$

Each M_k is a row in $\mathbb{F}^{4b_N+6b_G+3}$ encoding one of $\mathcal{V}$'s checks and $\vec{\pi}$ is a column in $\mathbb{F}^{4b_N+6b_G+3}$ comprising $\mathcal{P}$'s messages. ($4b_N+6b_G+3$ accounts for b_N rounds with cubic s_j, $2b_G$ rounds with quadratic s_j, and the final values v_0, v_1, and v_0v_1; §4.)

Now we can combine all of the linear equality checks encoded in Equation (5) into a single check, namely, by multiplying each row k by a random coefficient ρ_k and summing the rows.

Lemma 3. *For any $\vec{\pi} \in \mathbb{F}^\ell$, and any matrix $M \in \mathbb{F}^{n+1 \times \ell}$ with rows $M_1, \ldots, M_{n+1}$ for which Eq. (5) does not hold, then*

$$\Pr_\rho \left[\left\langle \left(\sum \rho_k \cdot M_k \right), \vec{\pi} \right\rangle = \rho_1 \cdot s_0 \right] \le 1/|\mathbb{F}|$$

Proof. Observe that $\langle (\sum \rho_k \cdot M_k), \vec{\pi} \rangle$ is a polynomial in $\rho_1, \ldots, \rho_{n+1}$ of total degree 1 (i.e., a linear function in $\rho_1, \ldots, \rho_{n+1}$). Call this linear polynomial ϕ. The coefficients of ϕ are the entries of $M \cdot \vec{\pi}$. Similarly, $\rho_1 \cdot s_0$ is a linear polynomial ψ in $\rho_1, \ldots, \rho_{n+1}$, whose coefficients are the entries of $[s_0, 0, \ldots, 0]$. Note that if Equation (5) does not hold, then ϕ and ψ are distinct polynomials, each of total degree 1. The lemma now follows from the Schwartz-Zippel lemma. $\square$

Putting the pieces together. Lemma 3 implies that, once $\mathcal{P}$ has committed to $\vec{\pi}$, it can use proof-of-dot-prod to convince $\mathcal{V}$ of the sum-check result in one shot. For soundness in Gir^{++}, however, $\mathcal{P}$ must commit to $c_{3,j}, c_{2,j}, c_{1,j}, c_{0,j}$ before the Verifier sends r_j. This means that $\mathcal{P}$ cannot send $\mathsf{Com}(\vec{\pi})$ all at once.

Instead, we observe that $\mathcal{P}$ can send the commitment to $\vec{\pi}$ *incrementally*, using one group element per round of the sum-check. That is, in each round of the sum-check protocol, $\mathcal{P}$ commits to a vector encoding the coefficients of that round's polynomial, and $\mathcal{V}$ responds with its random coin r_j. After $\mathcal{P}$ has committed to all of its messages for the sum-check, $\mathcal{P}$ and $\mathcal{V}$ engage in the protocol of Figure 1, which encodes $\mathcal{V}$'s checks for *all* rounds of the sum-check protocol at once. This protocol replaces $\mathcal{V}$'s checks in Step 2 of the protocol of Section 4.

Lemma 4. *The protocol of Figure 1 is a complete, honest-verifier perfect ZK argument, with witness-extended emulation under the discrete log assumption, that its inputs constitute an accepting sum-check relation: on input a commitment C_0, commitments $\{\alpha_j\}$ to polynomials $\{s_j\}$ in a sum-check invocation, rows $\{M_k\}$ of the matrix of Equation (5), and commitments $X = \mathsf{Com}(v_0)$, $Y = \mathsf{Com}(v_1)$, and Z, where $\{r_j\}$ are $\mathcal{V}$'s coins from the sum-check*

proof-of-sum-check(C_0, $\{\alpha_j\}$, $\{M_k\}$, X, Y, Z)

Inputs: $C_0 = \mathsf{Com}(s_0; r_{C_0})$.

$\{\alpha_j\}$ are all of $\mathcal{P}$'s messages from a sum-check invocation: at each round j of the sum-check protocol, $\mathcal{P}$ has sent

$$\alpha_j \leftarrow \mathsf{Com}((c_{3,j}, c_{2,j}, c_{1,j}, c_{0,j}); r_{\alpha_j})$$

$\{M_k\}$ is defined as in Equation (5) and Lemma 3. (These vectors encode $\mathcal{V}$'s random coins $\{r_j\}$ from the sum-check.)

$X = \mathsf{Com}(v_0; r_X)$, $Y = \mathsf{Com}(v_1; r_Y)$, $Z = \mathsf{Com}(v_0 v_1; r_Z)$.

Definitions: $n = b_N + 2b_G$; $\vec{\pi}$ is defined as in Equation (5); $\{\rho_k\}$ are chosen by $\mathcal{V}$ (see below); $\vec{J} = \sum \rho_k \cdot \vec{M}_k$; (J_X, J_Y, J_Z) are the last 3 elements of $\vec{J}$; $\vec{\pi}^*$ and $\vec{J}^*$ are all but the last three elements of $\vec{\pi}$ and $\vec{J}$, respectively.

1. $\mathcal{P}$ and $\mathcal{V}$ execute proof-of-product (§4) on $X, Y,$ and Z.

2. $\mathcal{P}$ picks $r_{\delta_1}, \ldots, r_{\delta_n} \in_R \mathbb{F}$ and $\vec{d} \in_R \mathbb{F}^{4b_N + 6b_G}$ where
   $\vec{d} = (d_{c_{3,1}}, d_{c_{2,1}}, d_{c_{1,1}}, d_{c_{0,1}}, \ldots, d_{c_{0,n-1}}, d_{c_{2,n}}, d_{c_{1,n}}, d_{c_{0,n}})$.
   $\mathcal{P}$ computes and sends
   $$\delta_j \leftarrow \mathsf{Com}((d_{c_{3,j}}, d_{c_{2,j}}, d_{c_{1,j}}, d_{c_{0,j}}); r_{\delta_j}), \quad j \in \{1, \ldots, n\}$$

3. $\mathcal{V}$ chooses and sends $\rho_1, \ldots, \rho_{n+1} \in_R \mathbb{F}$.

4. $\mathcal{P}$ picks $r_C \in_R \mathbb{F}$, then computes and sends
   $$C \leftarrow \mathsf{Com}(\langle \vec{J}^*, \vec{d} \rangle; r_C)$$

5. $\mathcal{V}$ chooses and sends challenge $c \in_R \mathbb{F}$.

6. $\mathcal{P}$ computes and sends $\vec{z} \leftarrow c \cdot \vec{\pi}^* + \vec{d}$,
   $$z_{\delta_j} \leftarrow c \cdot r_{\alpha_j} + r_{\delta_j}, \quad j \in \{1, \ldots, n\}, \text{ and}$$
   $$z_C \leftarrow c \cdot \left(\rho_1 r_{C_0} - J_X r_X - J_Y r_Y - J_Z r_Z\right) + r_C$$

7. $\mathcal{V}$ rejects unless the following holds, where we denote
   $\vec{z} = (z_{c_{3,1}}, z_{c_{2,1}}, z_{c_{1,1}}, z_{c_{0,1}}, \ldots, z_{c_{0,n-1}}, z_{c_{2,n}}, z_{c_{1,n}}, z_{c_{0,n}})$:
   $$\mathsf{Com}((z_{c_{3,j}}, z_{c_{2,j}}, z_{c_{1,j}}, z_{c_{0,j}}); z_{\delta_j}) \overset{?}{=} \alpha_j^c \odot \delta_j \quad j \in \{1, \ldots, n\}$$
   $$(C_0^{\rho_1} \odot X^{-J_X} \odot Y^{-J_Y} \odot Z^{-J_Z})^c \odot C \overset{?}{=} \mathsf{Com}(\langle \vec{J}^*, \vec{z} \rangle; z_C)$$

FIGURE 1—This protocol proves the statement derived by applying Lemma 3 to Equation (5), i.e., that the sum-check whose transcript is encoded in the protocol's inputs is accepting. Values corresponding to $c_{3,j}$ are elided for all sum-check rounds j having quadratic s_j.

and $n = b_N + 2b_G$, *the protocol proves that* $C_0 = \mathsf{Com}(s_1(0) + s_1(1))$; $s_j(0) + s_j(1) = s_{j-1}(r_{j-1}), j \in \{2, \ldots, n\}$; *and* $s_n(r_n) = Q_{\ldots,i}$ *evaluated with* v_0, v_1 *(per §3.2)*.

Lemma 4's proof is standard; we leave it to the full version [106, Appx. A.4]. Relative to Step 2 of Section 4, the protocol of Figure 1 reduces sum-check communication by $\approx 3\times$. It also reduces $\mathcal{P}$'s and $\mathcal{V}$'s cryptographic costs by $\approx 4\times$ and $\approx 5\times$, respectively.

6 Reducing the cost of the witness

In the protocol of Section 4, $\mathcal{P}$ sends a separate commitment to each element $w_1, \ldots, w_\ell$ of the witness w (§4, "Step 0."). This means that handling a circuit relation with $|w|$ witness elements requires a proof whose size is at least proportional to $|w|$. In this section, we describe a new commitment scheme for multilinear polynomials that reduces witness commitment size (and thus proof size) to sub-linear in $|w|$; it also reduces $\mathcal{V}$'s computation

cost to sub-linear in $|w|$ (§6.1). To begin, we require each sub-AC to have separate input and witness elements; we relax this restriction by introducing a *redistribution layer* that allows input and witness sharing among sub-ACs (§6.2).

6.1 A commitment scheme for multilinear polynomials

In Section 4, $\mathcal{V}$'s final step checks that $\mathcal{P}$'s commitments to w are consistent with its other messages by evaluating $\tilde{w}$ (the MLE of w; §3.2, "Multilinear extensions"). Zhang et al. [108] show, in the non-ZK setting, that $\mathcal{V}$ can outsource this evaluation to $\mathcal{P}$. We apply their idea to the ZK setting,[4] reducing communication and saving $\mathcal{V}$ computation, by devising a *polynomial commitment scheme* [61] tailored to multilinear polynomials. Informally, such schemes are hiding and binding (§3.1, Def. 4); they also allow the sender to evaluate a committed polynomial at any point and prove that the evaluation is consistent with the commitment.

Our commitment scheme builds on a matrix commitment idea due to Groth [52] and an inner-product argument due to Bünz et al. [30]. We begin by describing a simplified version of the scheme that gives $O(\sqrt{|w|})$ communication and $\mathcal{V}$ runtime; we then generalize this to $O(\mathsf{Sp})$ communication and $O(\mathsf{Ti})$ $\mathcal{V}$ runtime, $\mathsf{Ti} \geq \sqrt{|w|}$, $\mathsf{Sp} \cdot \mathsf{Ti} = |w|$. We assume WLOG for notational convenience that $2^\ell = |x| = |w|$.

Square-root commitment scheme. In its final check, $\mathcal{V}$ evaluates $\tilde{w}(r_1, \ldots, r_\ell)$ by computing a commitment to the dot product $\langle (w_0, \ldots, w_{2^\ell - 1}), (\chi_0, \ldots, \chi_{2^\ell - 1}) \rangle$ (Eq. (4), §4). Consider the following strawman protocol for computing this commitment: in Step 0 (§4), $\mathcal{P}$ sends one multi-commitment to w. Later, $\mathcal{P}$ sends a commitment ω, and $\mathcal{P}$ and $\mathcal{V}$ execute proof-of-dot-prod (§5) on $\mathsf{Com}(w)$, ω, and $(\chi_0, \ldots)$. This protocol convinces $\mathcal{V}$ that $\omega = \mathsf{Com}(\tilde{w}(\cdot))$, but does not reduce communication: proof-of-dot-prod requires $\mathcal{P}$ to send $O(|w|)$ messages (Appx. A.2).

To reduce communication, we exploit the structure of the polynomial $\tilde{w}$ and a matrix commitment due to Groth [52]. At a high level, this works as follows (details below). In Step 0, $\mathcal{P}$ encodes w as a matrix T, then sends commitments $\{T_k\}$ to the *rows* of T. Then, in the final step, $\mathcal{P}$ sends a commitment ω that it claims is to $\tilde{w}(r_1, \ldots, r_\ell)$; $\mathcal{V}$ uses $\{T_k\}$ to compute one multi-commitment T'; and $\mathcal{P}$ and $\mathcal{V}$ execute proof-of-dot-prod on T' and ω. In total, communication cost is $O(2^{\ell/2})$.

In more detail: T is the $2^{\ell/2} \times 2^{\ell/2}$ matrix whose column-major order is w, i.e., $T_{i+1,j+1} = w_{i+2^{\ell/2} \cdot j}$. Before defining T' and the proof-of-dot-prod invocation, we define

$$\check{\chi}_b = \prod_{k=1}^{\ell/2} \chi_{b_k}(r_k) \qquad \hat{\chi}_b = \prod_{k=\ell/2+1}^{\ell} \chi_{b_k}(r_k)$$

$$L = (\check{\chi}_0, \check{\chi}_1, \ldots, \check{\chi}_{2^{\ell/2}-1}) \qquad R = (\hat{\chi}_0, \hat{\chi}_{2^{\ell/2}}, \ldots, \hat{\chi}_{2^{\ell/2} \cdot (2^{\ell/2}-1)})$$

To compute T' from commitments $\{T_k\}$ to the rows of T, $\mathcal{V}$ evaluates L (in time $O(2^{\ell/2})$ [104, §3.3]) and uses it to compute

$$T' = \bigodot_{k=0}^{2^{\ell/2}-1} T_{k+1}^{\check{\chi}_k} = \mathsf{Com}(L \cdot T) \tag{6}$$

Finally, $\mathcal{P}$ sends a commitment ω and uses proof-of-dot-prod to convince $\mathcal{V}$ that the dot product of R with the vector committed in T' equals the value committed in ω.

---

[4]In concurrent and independent work, Zhang et al. extend to ZK [109]; see §2.

The above proves to $\mathcal{V}$ that $\omega = \mathsf{Com}(\tilde{w}(r_0, \ldots, r_\ell))$, as we now argue. For 1-indexed L and R, we have

$$L_{i+1} \cdot R_{j+1} = \check{\chi}_i \cdot \hat{\chi}_{2^{\ell/2} \cdot j} = \chi_{i+2^{\ell/2} \cdot j}$$

This is true because $\check{\chi}_b$ comprehends the lower $\ell/2$ bits of b, and $\hat{\chi}_b$ the upper $\ell/2$ bits. Then by the definition of T, we have

$$L \cdot T \cdot R^{\mathrm{T}} = \sum_{i=0}^{2^{\ell/2}-1} \sum_{j=0}^{2^{\ell/2}-1} T_{i+1,j+1} \cdot L_{i+1} \cdot R_{j+1}$$

$$= \sum_{i=0}^{2^{\ell/2}-1} \sum_{j=0}^{2^{\ell/2}-1} w_{i+2^{\ell/2} \cdot j} \cdot \chi_{i+2^{\ell/2} \cdot j} \; = \sum_{k=0}^{2^\ell-1} w_k \cdot \chi_k$$

If $\mathcal{V}$ accepts $\mathcal{P}$'s proof-of-dot-prod on T', ω, and R, then by Equation (6), $\omega = \mathsf{Com}(L \cdot T \cdot R^{\mathrm{T}}) = \mathsf{Com}(\sum_{k=0}^{2^\ell-1} w_k \cdot \chi_k)$, which equals $\mathsf{Com}(\tilde{w}(r_0, \ldots, r_\ell))$ (Eq. (4), §4) as claimed.

In total, communication is $\mathrm{O}(2^{\ell/2})$ (for $\{T_k\}$ plus the proof-of-dot-prod invocation), and $\mathcal{V}$'s computational cost is $\mathrm{O}(2^{\ell/2})$ (for computing L, R, and T', and executing proof-of-dot-prod).

Reducing the cost of proof-of-dot-prod. In the above protocol, proof-of-dot-prod establishes a lower bound on communication cost. To reduce proof-of-dot-prod's cost, we use an idea due to Bünz et al. [30], who give a dot-product protocol that has cost logarithmic in the length of the vectors. Their protocol works over two committed vectors; we require one that works over one committed and one public vector. In Appendix A.3, we adapt their protocol to the syntax of proof-of-dot-prod; we refer to the result as $\mathrm{proof}_{\log}$-of-dot-prod. Whereas proof-of-dot-prod requires $\mathcal{P}$ to send $4+n$ elements for vectors of length n, $\mathrm{proof}_{\log}$-of-dot-prod requires only $4+2\log n$. In both protocols, $\mathcal{V}$'s computational cost is dominated by a multi-exponentiation [83] of length n.

The full commitment scheme differs from the square-root one in that $\mathcal{P}$ and $\mathcal{V}$ invoke $\mathrm{proof}_{\log}$-of-dot-prod (rather than proof-of-dot-prod) on T', R, and ω. For $T, L, R, \check{\chi}_b, \hat{\chi}_b$ as defined above, $\mathcal{P}$ sends $4+2^{\ell/2}+2\log 2^{\ell/2}$ elements, and $\mathcal{V}$'s runtime is dominated by two multi-exponentiations of length $2^{\ell/2}$, one to compute T' and the other to execute $\mathrm{proof}_{\log}$-of-dot-prod. This gives the same asymptotics as the square-root scheme with $\approx 2\times$ less communication (but with $\approx 3\times$ more computation for $\mathcal{P}$).

More importantly, $\mathrm{proof}_{\log}$-of-dot-prod gives the freedom to reduce communication in exchange for increased $\mathcal{V}$ runtime. For a parameter ι, we redefine T to be the $2^{\ell/\iota} \times 2^{\ell-\ell/\iota}$ matrix whose column-major order is w; redefine $\check{\chi}_b$ to comprehend the lower ℓ/ι bits of b, and $\hat{\chi}_b$ the upper $\ell - \ell/\iota$ bits; and redefine

$$L = (\check{\chi}_0, \check{\chi}_1, \ldots, \check{\chi}_{2^{\ell/\iota}-1}) \quad R = (\hat{\chi}_0, \hat{\chi}_{2^{\ell/\iota}}, \ldots, \hat{\chi}_{2^{\ell/\iota} \cdot (2^{\ell-\ell/\iota}-1)})$$

T has $2^{\ell/\iota}$ rows and T' is a vector of $2^{\ell-\ell/\iota}$ elements, so $\mathcal{P}$ sends $2^{\ell/\iota}$ commitments in Step 0 and $4+\log 2^{\ell-\ell/\iota}$ elements for $\mathrm{proof}_{\log}$-of-dot-prod, which is $\mathrm{O}(2^{\ell/\iota})$ in total. Computing T' costs $\mathcal{V}$ one multi-exponentiation of length $2^{\ell/\iota}$, and executing $\mathrm{proof}_{\log}$-of-dot-prod costs one of length $2^{\ell-\ell/\iota}$, which is $\mathrm{O}(2^{\ell/\iota}+2^{\ell-\ell/\iota})$ in total. Since this is at least $\mathrm{O}(2^{\ell/2})$, $\mathcal{V}$'s runtime is at least $\mathrm{O}(\sqrt{|w|})$. We formalize immediately below.

Lemma 5. *Suppose WLOG that $w \in \mathbb{F}^{2^{\iota \cdot \ell'}}$ for $\iota \geq 2$, and that $\mathcal{P}$ commits to w as described above using $2^{\ell'} = |w|^{1/\iota}$ multi-commitments. Then for any $(r_1, \ldots, r_{\iota \cdot \ell'})$, $\mathcal{P}$ can send a commitment ω and argue that it commits to $\tilde{w}(r_1, \ldots, r_{\iota \cdot \ell'})$ in communication*

$\mathrm{O}(|w|^{1/\iota})$, *where $\mathcal{V}$ runs in $\mathrm{O}(|w|^{(\iota-1)/\iota})$ steps. This is a complete, honest-verifier perfect zero-knowledge argument with witness-extended emulation under the discrete log assumption.*

Completeness and ZK follow from the analysis in Appendix A.3. We leave analysis of witness-extended emulation to the full version [106, Appx. A.5] for space reasons. We have described this protocol in terms of the multilinear extension of w, but it generalizes to any multilinear polynomial f using the fact that T comprises the evaluations of f at all binary inputs.

6.2 Sharing witness elements in the data-parallel setting

We have thus far regarded the computation as having one large input and one large witness. When evaluating a data-parallel computation, this means that the sub-ACs' inputs must be disjoint slices of the full input (and similarly for the witness). However, this is not sufficient in many cases of interest.

Consider a case where $\mathcal{P}$ wants to convince $\mathcal{V}$ that it knows leaves of a Merkle tree corresponding to a supplied root. Verifying a witness with M leaves requires $2M-1$ invocations of a hash function. We encode this as a computation with $2M-1$ sub-ACs laid side-by-side, each encoding the hash function.[5] Then, for sub-AC$_b$ processing sub-AC$_a$'s output, $\mathcal{P}$ supplies the purported output to both, and sub-AC$_a$ just *checks* that value and outputs a bit indicating correctness. This is necessary for zero-knowledge: all AC outputs are public, whereas sub-AC$_a$'s output (an intermediate value in the computation) must not be revealed to $\mathcal{V}$.

This arrangement requires sub-ACs to share witness elements—but duplicating entries in the matrix T (§6.1) is not a solution, because $\mathcal{V}$ cannot detect if a cheating $\mathcal{P}$ produces T that gives different values to different sub-ACs. One possibility is a hybrid vector-scalar scheme: $\mathcal{P}$ supplies scalar commitments for each shared witness element and matrix commitment $\{T_k\}$ for the rest. Then, for a scalar commitment δ, $\mathcal{V}$ "injects" the committed value into input index b by multiplying the commitment to $\tilde{V}_d(r', r_\upsilon)$ (§4, "Final step") by $\delta^{-r_0 \cdot \chi_b}$.[6] (In contrast, the protocol of Section 6.1 maps each entry of T to a fixed input index.)

This approach works when the number of shared witness elements is small, but it is inefficient when there are many shared elements: each shared element requires a separate commitment and proof-of-opening invocation. For such cases, we enable sharing of witness elements by modifying the arithmetic circuit encoding the NP relation. Specifically, after constructing a data-parallel AC corresponding to the computation, we add one non-data-parallel *redistribution layer* (RDL) whose inputs are the full input and witness, and whose outputs feed the input layers of each sub-AC. Since the RDL is not data parallel, there are no restrictions on how its inputs connect to its outputs, meaning

---

[5] The sub-ACs could instead be arranged sequentially. This would avoid the issues described in this subsection, but would dramatically increase circuit depth, and thus the proof length and associated costs when applying our argument.

[6] In fact, this approach works generally for computations over values to which $\mathcal{V}$ holds a commitment whose opening $\mathcal{P}$ knows. It also applies to committed *vectors*: if $\mathcal{V}$ holds a commitment $\xi = \mathsf{Com}(\vec{x})$, it can inject the committed values into a list of indices $(b_1, \ldots)$ as follows: $\mathcal{P}$ produces a commitment δ and proves to $\mathcal{V}$ that it commits to $\mathsf{Com}(\langle (x_1, \ldots), (\chi_{b_1}, \ldots) \rangle)$ with $\mathrm{proof}_{\log}$-of-dot-prod; then $\mathcal{V}$ multiplies $\mathsf{Com}(\tilde{V}_d(r', r_\upsilon))$ by δ^{-r_0}. This approach requires more communication and $\mathcal{V}$ computation than the protocol of Section 6.1, because it does not assume any particular structure for $(b_1, \ldots)$.

that $\mathcal{V}$ can use it to ensure that the same witness element feeds multiple sub-ACs: the sum-check protocol forces $\mathcal{P}$ to respect the wiring of the RDL, so $\mathcal{P}$ cannot equivocate about w.

Moreover, since the RDL only "re-wires" its inputs, the sum-check invocation corresponding to this layer of the AC can be optimized to require fewer rounds and a simplified final check. Observe that the redistribution layer only requires one-input "pass" gates that copy their input to their output. Thus, following a simplification of the CMT protocol [35, 98], we have that

$$\tilde{V}_{d-1}(q', q) = \sum_{h \in \{0,1\}^{\log(|m|)}} \mathrm{p\tilde{a}ss}((q', q), h) \cdot \tilde{V}_d(h)$$

where $\mathrm{p\tilde{a}ss}((q', q), h)$ is the MLE of a wiring predicate (§3.2) that is 1 when the RDL connects from the AC input with index h to input q in sub-AC q', and 0 otherwise. A sum-check over

$$\mathrm{RDL}_{(q', q)}(h) = \mathrm{p\tilde{a}ss}((q', q), h) \cdot \tilde{V}_d(h)$$

requires $\log(|m|) = \log(|x| + |w|)$ rounds, at the end of which $\mathcal{V}$ evaluates $\mathrm{RDL}_{(q', q)}$ at a random point. This requires $\mathcal{V}$ to evaluate $\mathrm{p\tilde{a}ss}$, but in contrast to $P_{...,i}$ or $Q_{...,i}$ (§3.2), it only requires *one* evaluation of $\tilde{V}_d$, which $\mathcal{V}$ can check (via the protocol of §6.1) without invoking a mini-protocol (§3.2).

By a standard analysis [35], $\mathcal{P}$'s costs are $O(NG \log |m|)$; $\mathcal{V}$'s primary cost related to the RDL is evaluating $\mathrm{p\tilde{a}ss}$ at one point, which costs $O(|m| + NG)$ via known techniques [104, §3.3]. We formalize in Theorem 6 (§7).

7 Hyrax: a zkSNARK based on Gir^{++}

We refer to the honest-verifier PZK argument obtained by applying the refinements of Sections 5 and 6 to the protocol of Section 4 as Hyrax-I; pseudocode is given in Appendix B. Since Hyrax-I is a public-coin protocol, we apply the Fiat-Shamir heuristic [41] to produce a zkSNARK that we call Hyrax whose properties we now formalize:

Theorem 6. *Let $C(\cdot, \cdot)$ be a layered AC of fan-in two, consisting of N identical sub-computations, each having d layers whose width is at most G. Under the discrete log assumption in the random oracle model, for every* Sp, Ti *with* Ti $\geq \sqrt{|w|}$ *and* Sp $\cdot$ Ti $=$ $|w|$, *there exists a perfectly complete, perfect zero-knowledge, non-interactive argument with witness-extended emulation for the NP relation "$\exists w$ such that $C(x, w) = y$." $\mathcal{V}$ runs in time* $O(|x| + |y| + dG + (\mathrm{Ti} + d \log(NG)) \cdot \kappa)$ *for κ a bound on the time to compute a commitment; when using an RDL (§6.2), $\mathcal{V}$ incurs an additional* $O(|x| + |w| + NG)$ *cost. $\mathcal{P}$'s messages have size* $O((\mathrm{Sp} + d \log(NG)) \cdot \lambda)$ *for λ a security parameter.*

We leave proof to the full version [106, Appx. B].

Implementation. Our implementation of Hyrax is based on Giraffe's code [81, 104]. It uses Pedersen commitments (Appx. A) in an elliptic curve group of order $q_{\mathcal{G}}$ and works with ACs over $\mathbb{F}_{q_{\mathcal{G}}}$. We instantiate the random oracle with SHA-256.

The prover takes as input a high-level description of an AC (in the format produced by Giraffe's C compiler), the public inputs, and an auxiliary executable that generates the witness from the public inputs; the prover's output is a proof. The verifier takes as input the same computation description and public inputs plus the proof, and outputs "accept" or "reject."

We implement Gir^{++}, the techniques of Sections 5 and 6, the random oracle, and proof serialization and deserialization

by adding 2800 lines of Python and 300 lines of C to the Giraffe code. We also implemented a library for fast multi-exponentiation comprising 750 lines of C that uses the MIRACL Crypto SDK [74] for elliptic curve operations and selects between Straus's [95] and Pippenger's [19, 83] methods, depending on the problem size. Our library supports Curve25519 [18], M221, M191, and M159 [2]. Python code calls this library via CFFI [31]. We produce random group elements by hashing, implemented in 200 lines of Sage [88] adapted from a script by Samuel Neves [2].

We have released full source code [58].

8 Evaluation

In this section we ask:
- How does Hyrax compare to several baseline systems, considering proof size and $\mathcal{V}$ and $\mathcal{P}$ execution time?
- How do Hyrax's refinements (§5–6) improve its costs?
- What is the overall effect of trading greater witness-related $\mathcal{V}$ computation for smaller witness commitments (§6.1)?

A careful comparison of built systems shows that, even for modest problem sizes, Hyrax's proofs are smaller than all but the most computationally costly of the baselines; and that its $\mathcal{V}$ and $\mathcal{P}$ execution times are each faster than three of five baselines. We also find that Hyrax's refinements yield multiple-orders-of-magnitude savings in proof size and $\mathcal{V}$ time, and a small constant savings in $\mathcal{P}$ time. Finally, we find that tuning the witness commitment costs gives much smaller proofs, with little effect on total $\mathcal{V}$ time for a computation using an RDL (§6.2).

8.1 Comparison with prior work

Baselines. We compare Hyrax with five state-of-the-art zero-knowledge argument systems with similar properties, detailed below. We also consider *Hyrax-naive*, which implements the protocol of Section 4 without our refinements (§5–6). We do not compare to systems that require trusted setup (see §2, second paragraph), but we discuss them briefly in Section 8.3.

Like Hyrax (and Hyrax-naive), two of the baselines rely on elliptic curve primitives; but their existing implementations use a different elliptic curve than Hyrax. To evaluate like-for-like, we re-implemented them using the Python scaffolding, C cryptographic library, and elliptic curves that Hyrax uses (§7).

The other three baselines do not use elliptic curves, so some mismatch in implementations is unavoidable. For those systems, we used existing implementations written in C or C++.

• *BCCGP-sqrt* is the square-root-communication argument due to Bootle et al. [24]. We implemented this protocol using Hyrax's libraries, as described above. In addition, this protocol uses polynomial multiplication, for which we used NTL [94]. Finally, we wrote a compiler that converts from Hyrax's AC description format to the required constraint format, with rudimentary optimizations like constant folding and common subexpression elimination. Our implementation comprises 1200 lines of Python and 160 lines of C, which we include in our released code [58].
• *Bulletproofs* is the argument due to Bünz et al. [30] (we also adapted the inner-product argument from this work in §6). We implemented this protocol in 300 lines of Python on top of our BCCGP-sqrt code, which we also include in our release.
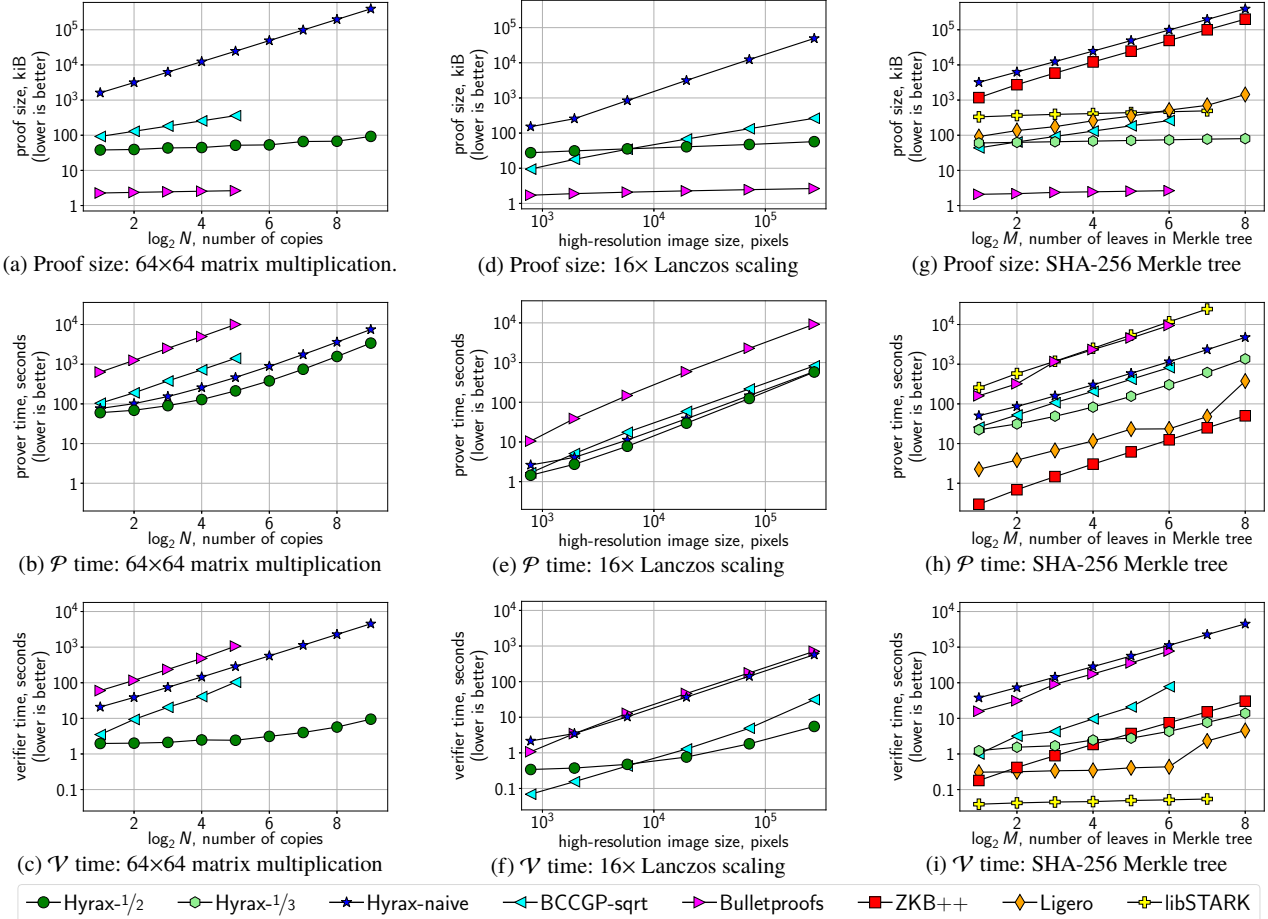
(a) Proof size: 64×64 matrix multiplication.

(d) Proof size: 16× Lanczos scaling

(g) Proof size: SHA-256 Merkle tree

(b) $\mathcal{P}$ time: 64×64 matrix multiplication

(e) $\mathcal{P}$ time: 16× Lanczos scaling

(h) $\mathcal{P}$ time: SHA-256 Merkle tree

(c) $\mathcal{V}$ time: 64×64 matrix multiplication

(f) $\mathcal{V}$ time: 16× Lanczos scaling

(i) $\mathcal{V}$ time: SHA-256 Merkle tree

— Hyrax-$1/2$ — Hyrax-$1/3$ —★— Hyrax-naive   —◁— BCCGP-sqrt   —▶— Bulletproofs   —■— ZKB++   —◆— Ligero   —+— libSTARK

FIGURE 2—Comparison of concrete costs between the baseline systems and Hyrax (§8.1). Hyrax-$1/2$ is Hyrax where $\mathcal{P}$'s witness commitments have size $|w|^{1/2}$, and likewise Hyrax-$1/3$ has commitments of size $|w|^{1/3}$ (§6.1). Hyrax-naive is Hyrax without our refinements (§5–6). BCCGP-sqrt [24], Bulletproofs [30], ZKB++ [32], Ligero [1], and libSTARK [11] are prior work. Where the BCCGP-sqrt, Bulletproofs, and libSTARK data are truncated, their provers exceeded available RAM (§8.1). We evaluate ZKB++, Ligero, and libSTARK only on Merkle trees (we discuss in §8.3).

• *Ligero* [1]: we report on the authors' C++ implementation.
• *ZKB++* [32]: we report runtime of the C implementation of ZKBoo [47, 111]; per the ZKB++ authors, these systems have similar performance [32, §3.2].[7] We report extrapolated proof sizes (which are linear with AC size) from ZKB++ [32, §3.2.1].
• *libSTARK* [11]: we report on the authors' C++ implementation [67] and SHA-256 primitive [11, Fig. 4], which we adapt to the Merkle tree benchmark (described below).

This implementation supports multi-threading, but we restrict it to a single thread for consistency with the other baselines and to focus on total prover work; we discuss in Section 8.3.

Benchmarks. We evaluate Hyrax, Hyrax-naive, BCCGP-sqrt, and Bulletproofs on all benchmarks below, but Ligero, ZKB++, and libSTARK only on Merkle trees; we discuss in Section 8.3.
• *Matrix factoring* (i.e., matrix multiplication) proves to $\mathcal{V}$ that $\mathcal{P}$ knows two matrices whose product equals the public input. We evaluate on 16×16, 32×32, 64×64, and 128×128 matrices, and for each we vary N, the number of parallel executions.

• *Image scaling* establishes that $\mathcal{V}$'s input, a low-resolution image, is a scaled version of a high-resolution image that $\mathcal{P}$ knows. For scaling, we use Lanczos resampling [100], a standard image transformation in which each output pixel is the result of convolving a two-dimensional windowed sinc function [77] with the input image. We evaluate on 4×, 16×, 64×, and 256× scaling, varying the number of pixels.

This is a data-parallel computation where each sub-AC evaluates one pixel of the low-resolution image, but because of the windowed sinc function, sub-ACs for adjacent pixels must share inputs from the high-resolution image. To accommodate this, we use a redistribution layer (RDL; §6.2).
• *Merkle tree* proves to $\mathcal{V}$ that $\mathcal{P}$ knows an assignment to the leaves of a Merkle tree [72] corresponding to a root that $\mathcal{V}$ provides [23].[8] We use SHA-256 for the hash, varying the number of leaves in the tree; we implement a data-parallel computation in which each sub-computation is one invocation of SHA-256;

---

[7]We run ZKBoo because there is no standalone ZKB++ implementation that can run our benchmarks, only one tailored to the Picnic signature scheme [82].

[8]In related applications (e.g., [107]), $\mathcal{P}$ convinces $\mathcal{V}$ that it knows a *path* from a supplied Merkle root to a leaf. For these systems, a path of length $2M-1$ has essentially the same cost as an M-leaf Merkle tree. We evaluate the full-tree benchmark because it demonstrates a wider range of computation sizes.

and we connect outputs at one level of the tree to inputs at the next level using an RDL. For M leaves, the benchmark comprises $2M-1$ sub-computations.

To implement SHA-256 efficiently in an arithmetic circuit, we use an approach from prior work [12] for efficient addition modulo 2^{32}. We describe the approach, and an optimization that may be of independent interest, in the full version [106, Appx. C].

Testbed. We run experiments on Amazon EC2 [3]. For Hyrax, Hyrax-naive, Ligero, and ZKB++, we use c3.4xlarge instances (30 GiB of RAM, 8 Xeon E5-2680v2 cores, 2 threads per core, 2.8 GHz). The BCCGP-sqrt, Bulletproofs, and libSTARK provers are memory intensive ("$\mathcal{P}$ cost," below), so for these we use c3.8xlarge instances (60 GiB of RAM, 16 cores at 2.8 GHz). Only RAM is relevant because we run all tests single threaded.

All testbed machines run Debian GNU/Linux 9 [38]. We run all Python code using PyPy [86], a fast JIT-compiling interpreter.

Security parameters. For Hyrax, Hyrax-naive, BCCGP-sqrt, and Bulletproofs, $\mathcal{G}$ is M191 [2], an elliptic curve over a base field modulo $2^{191}-19$ with a subgroup of order $q_{\mathcal{G}} = 2^{188}+2^{93}+\ldots$, giving $\approx$90-bit security. We run Ligero, ZKB++, and libSTARK at 2^{-80} soundness error.[9] ACs are over $\mathbb{F}_{q_{\mathcal{G}}}$ and group elements and scalars are 24 bytes, except that Ligero and libSTARK work over smaller fields and ZKB++ works over Boolean circuits.

Method. For each benchmark, we construct a set of arithmetic circuits (and, for image scaling and Merkle trees, RDLs) for a range of computation sizes. We then run each system's prover, feeding the resulting proof into its verifier. We record proof size, and measure time using the high-resolution system clock.

For matrix factoring and image scaling, we set Hyrax's communication and $\mathcal{V}$ runtime to $|w|^{1/2}$ (§6.1). For Merkle trees, we optimize proof size versus $\mathcal{V}$ runtime by setting witness-related communication to $|w|^{1/3}$ and $\mathcal{V}$ runtime to $|w|^{2/3}$; we explore the effect of this setting in Section 8.2.

Results. Figure 2 compares costs for the benchmarks. For matrix factoring and image scaling we show only 64×64 matrices and 16× scaling, respectively; other values give similar results. For an AC C, M denotes the number of multiplication gates.

Proof size (Figs. 2a, 2d, 2g):
• Hyrax has much larger proofs than Bulletproofs, both asymptotically and concretely.
• Hyrax's proofs are smaller than BCCGP-sqrt's when the cost of the witness commitment dominates the cost of $\mathcal{P}$'s messages in Gir^{++} (i.e., for large enough computations). Specifically, Hyrax's cost tracks $|w|^{1/2}$ ($|w|^{1/3}$ for Merkle trees; Fig. 2g), while BCCGP-sqrt's tracks $M^{1/2}$. Thus, on matrix factoring (where $|w| \ll M$) Hyrax has much smaller proofs.
• Hyrax's Merkle tree proofs are asymptotically and concretely smaller than Ligero's: the latter's cost tracks $|C|^{1/2}$.
• libSTARK's proof size is asymptotically smaller than Hyrax's, but its proofs are concretely $\approx$5× larger at these problem sizes.
• ZKB++'s cost is linear in the number of AND gates, and Hyrax-naive's cost tracks $|w|$; both are large.

$\mathcal{P}$ *cost* (Figs. 2b, 2e, 2h):
• BCCGP-sqrt and Bulletproofs require a number of cryptographic operations proportional to M. Hyrax has lower $\mathcal{P}$ time than these systems because it uses cryptographic operations only for $\mathcal{P}$'s messages in Gir^{++} and for w (§4–§6).
• The provers in both BCCGP-sqrt and Bulletproofs ran out of memory for the largest benchmarks (Figs. 2b and 2h) despite having twice as much RAM as Hyrax ("Testbed," above). This is because they operate, roughly speaking, over all wire values in the AC at once. In contrast, Hyrax's $\mathcal{P}$ works layer-by-layer (§3.2).[10]
• Hyrax's $\mathcal{P}$ is more expensive than either ZKB++'s or Ligero's, because those systems do not use any public-key cryptography.
• While Ligero's $\mathcal{P}$ is asymptotically more costly than Hyrax's $\mathcal{P}$, this is not apparent at the problem sizes we consider.
• libSTARK's $\mathcal{P}$ is 12–40× more expensive than Hyrax's for these problem sizes. It is also memory intensive: for the largest problem, it exceeded available RAM despite having twice as much as Hyrax ("Testbed," above).
• Hyrax's refinements compared to Hyrax-naive (§5–6) yield a constant factor lower $\mathcal{P}$ cost, at most $\approx$3×.

$\mathcal{V}$ *time* (Fig. 2c, 2f, 2i):
• For matrix factoring, Hyrax's $\mathcal{V}$ bottleneck is sum-check invocations for small N, and $\tilde{V}_y$ evaluation for large N (§3.2). The RDL (§6.2) dominates $\mathcal{V}$'s costs in the other two benchmarks.
• Hyrax's $\mathcal{V}$ cost is lower than BCCGP-sqrt for large enough problems: the latter requires O(M) field operations.
• Hyrax's $\mathcal{V}$ cost is much less than Bulletproofs's: the latter requires a multi-exponentiation of length $2M$ (which can be computed using O($M/\log M$) cryptographic operations [83]).
• ZKB++ has verification cost linear in the problem size, so Hyrax wins on large enough problems.
• Ligero's $\mathcal{V}$ amortizes its bottleneck computation over repeated SHA-256 instances [1, §5.4], so over this range of problem sizes it has sublinear scaling and concretely fast verification time.
• libSTARK's $\mathcal{V}$ has the best asymptotics among all systems and extremely low concrete costs.
• Hyrax-naive requires cryptographic operations proportional to $|w|$; Hyrax's refinements give more than 100× savings.

8.2 Effect of trading $\mathcal{V}$ runtime for smaller proofs

Method. We run the Merkle tree benchmark using the same setup as in Section 8.1, except that we vary the size of $\mathcal{P}$'s witness commitment (§6.1). We experiment with commitments of size $\log|w|$, $|w|^{1/3}$, and $|w|^{1/2}$. $\mathcal{V}$'s witness-related work at these three settings is O($|w|$), O($|w|^{2/3}$), and O($|w|^{1/2}$), respectively.

Results. Figure 3 shows proof size and runtime for the specified commitment sizes. For Hyrax-1/2, proof sizes are large but $\mathcal{P}$ and $\mathcal{V}$ runtimes are small; Hyrax-log is the opposite. Hyrax-1/3 has similar runtimes to Hyrax-1/2: $\mathcal{P}$'s costs are dominated by Gir^{++}, $\mathcal{V}$'s by the RDL (§6.2). Meanwhile, its proof sizes are not much larger than Hyrax-log, because the Gir^{++}-related proof costs are the same in both cases, and because the constants hidden in the asymptotic notation mean that the log and cube-root protocols

---

[9]ZKB++, Ligero, and libSTARK give statistical security in the random oracle model, while the other systems make computational assumptions; this makes direct comparison difficult. We have chosen security parameters to give all systems roughly equivalent cost to prove a false statement.

[10]It is probably possible to engineer the BCCGP-sqrt and Bulletproofs provers to reduce memory requirements, e.g., by streaming from disk. We attempted a standard approach—paging memory to an array of fast SSDs—but this caused thrashing and dramatically worsened runtimes.
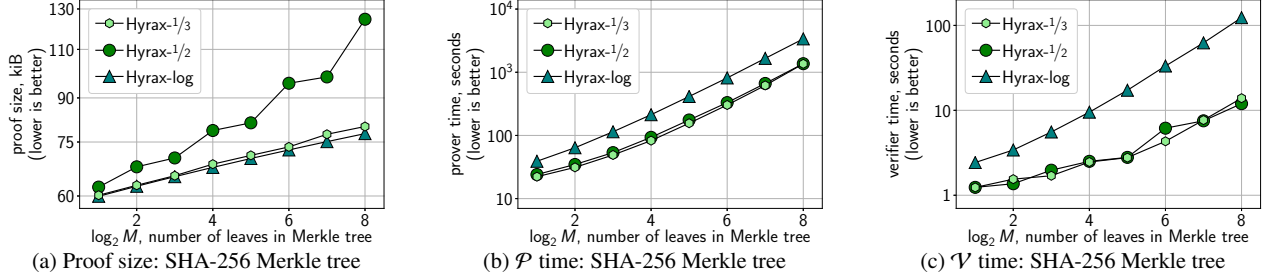
| (a) Proof size: SHA-256 Merkle tree | (b) $\mathcal{P}$ time: SHA-256 Merkle tree | (c) $\mathcal{V}$ time: SHA-256 Merkle tree |

FIGURE 3—Proof size and $\mathcal{P}$ and $\mathcal{V}$ runtime for different sizes of $\mathcal{P}$'s witness commitment (§6.1; §8.2). Hyrax-1/2 has commitment size $|w|^{1/2}$, Hyrax-1/3 has commitment size $|w|^{1/3}$, and Hyrax-log has commitment size $\log |w|$. Hyrax-1/2 gives the largest proofs but has the fastest runtimes. Hyrax-log gives the smallest proofs but has the longest runtimes. Hyrax-1/3 gets essentially the best of both for this application.

have similar concrete costs at these problem sizes. In other words, Hyrax-1/3 gets very nearly the best of both worlds.

8.3 Discussion

Our results show that Hyrax is competitive with the baselines, and that the refinements of Sections 5 and 6 give substantial improvements. Hyrax gives smaller proofs than all but Bulletproofs, which pays for its smaller proofs with very high computational costs. Meanwhile, for problem sizes of practical interest, only Ligero is faster for both $\mathcal{P}$ and $\mathcal{V}$; ZKB++ has faster $\mathcal{P}$ but often slower $\mathcal{V}$; libSTARK has faster $\mathcal{V}$ but much slower $\mathcal{P}$; and all three systems produce larger proofs than Hyrax.

On the other hand, there are several limitations to this analysis. First, because Gir++ is geared to data-parallel computations (§3.2; Thm. 1), Hyrax is competitive with prior work primarily when computations contain sufficient parallelism or are amenable to batching; this is evident in the way Hyrax's performance relative to the baselines improves as parallelism increases in Figure 2. While an RDL (§6.2) lets Hyrax take advantage of parallelism within one computation (as it did in the Merkle tree and image scaling benchmarks), not all applications fit these paradigms. Moreover, the RDL is asymptotically and concretely costly for $\mathcal{V}$; eliminating this bottleneck is future work.

Second, we compare ZKB++, Ligero, and libSTARK only on the SHA-256 Merkle tree benchmark. This makes sense for ZKB++ because it is geared to Boolean circuits, where SHA-256 is a natural benchmark; similarly, Ligero's primary evaluation is on SHA-256 [1, §6]. For libSTARK, however, a hash function that is more efficient in $\mathbb{F}_{2^{64}}$ would improve performance [11, Fig. 4]; future work is to compare Hyrax and all baselines on Merkle trees using hash functions tailored to each system (e.g., [11, Appx. E; 15, §5.2; 29, §3.2]). Furthermore, Ligero and libSTARK can in principle work over large fields, but the current implementations do not [11, 101], so we could not evaluate on matrix factoring or image scaling; future work is to do so.[11]

Third, our comparison does not consider multi-threaded performance because, to our knowledge, libSTARK is the only baseline with a multi-threaded implementation [11, 67]. Prior work [99, 103, 104] suggests that Gir++ is highly parallelizable; exploring this in Hyrax is future work.

Finally, our comparison does not consider argument systems like libsnark [16, 66] that require trusted setup and non-standard, non-falsifiable assumptions (§2, paragraph 2); Hyrax's goal is to avoid these requirements. Ignoring this, Hyrax's proofs are bigger: libsnark's proofs are a constant ≈300-bytes, independent of the AC C. Hyrax's $\mathcal{P}$ cost is concretely and asymptotically smaller: libsnark has a logarithmic overhead in $|C|$, and it requires cryptographic operations per AC gate, while Hyrax's $\mathcal{P}$ is essentially linear in computation size and requires cryptographic operations only for $\mathcal{P}$'s Gir++ messages and for w (§4–§6). For $\mathcal{V}$, libsnark's offline setup is very expensive [105, §5.4], and it must be performed by $\mathcal{V}$ or someone $\mathcal{V}$ trusts; but libsnark's online $\mathcal{V}$ costs are essentially always cheaper than Hyrax's (and roughly comparable to libSTARK's, in practice).

9 Conclusion

We have described a succinct zero-knowledge argument for NP with no trusted setup and low concrete cost for both the prover and the verifier, based on standard cryptographic assumptions. This scheme is practical because it tightly integrates three components: a state-of-the-art interactive proof (IP), which we tweak to reduce communication complexity; a highly optimized transformation from IPs to zero-knowledge arguments following the approach of Ben-Or et al. [8] and Cramer and Damgård [37]; and a new cryptographic commitment scheme tailored to multilinear polynomials that adapts prior work [30, 52] to allow a sender to commit to a $\log G$-variate multilinear polynomial and later to open it at one point, with $O(G^{1/\iota})$ total communication and $O(G^{(\iota-1)/\iota})$ receiver runtime for any $\iota \geq 2$. A careful comparison with prior work shows that our argument system is competitive on both proof size and computational costs. Key future work is to further reduce proof size without increasing verifier runtime.

More broadly, ours and other recent work [108–110] suggest that the applicability of the GKR interactive proof [49] has been underestimated. In particular, GKR seemingly requires deterministic arithmetic circuits, and saves work for the verifier (relative to computing the circuit) only when those circuits have low depth. Zhang et al. sidestep these issues, extending GKR to non-deterministic, low-depth computations [108] and more recently to arbitrary RAM programs [110], in both cases saving work asymptotically for the verifier. But even those enhanced protocols fall short of state-of-the-art work-saving zkSNARKs [4, 5, 12, 13, 15, 16, 29, 34, 36, 39, 42–44, 64, 76, 79, 105], because

---

[11]We note that, since Hyrax's proof size is primarily due to witness size $|w|$ rather than arithmetic circuit size $|C|$, we expect it to outperform Ligero on applications like matrix factoring where $|w| \ll |C|$.

they fail to address zero-knowledge applications. This work (and concurrent work by Zhang et al. [109]) closes that gap—and, in our view, attests to the power and versatility of the GKR protocol.

We have released Hyrax's source code and our BCCGP-sqrt and Bulletproofs implementations as open-source software [58].

Acknowledgments

References

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

[2] D. F. Aranha, P. S. L. M. Barreto, G. C. C. F. Pereira, and J. E. Ricardini. A note on high-security general-purpose elliptic curves. Cryptology ePrint Archive, Report 2013/647, 2013.

[3] AWS EC2. https://aws.amazon.com/ec2/instance-types/.

[4] M. Backes, M. Barbosa, D. Fiore, and R. M. Reischuk. ADSNARK: Nearly practical and privacy-preserving proofs on authenticated data. In *IEEE S&P*, May 2015.

[5] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *ACM CCS*, Nov. 2013.

[6] S. Bayer and J. Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *EUROCRYPT*, Apr. 2012.

[7] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *ACM CCS*, Nov. 1993.

[8] 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*, Aug. 1990.

[9] E. Ben-Sasson, I. Ben-Tov, A. Chiesa, A. Gabizon, D. Genkin, M. Hamilis, E. Pergament, M. Riabzev, M. Silberstein, E. Tromer, and M. Virza. Computational integrity with a public random string from quasi-linear PCPs. In *EUROCRYPT*, Apr. 2017.

[10] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Fast Reed-Solomon interactive oracle proofs of proximity. *Electronic Colloquium on Computational Complexity (ECCC)*, 24:134, 2017.

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

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

[13] 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.

[14] E. Ben-Sasson, A. Chiesa, and N. Spooner. Interactive oracle proofs. In *IACR TCC*, Oct. 2016.

[15] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, Aug. 2014.

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

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

[18] D. J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *PKC*, Apr. 2006.

[19] D. J. Bernstein, J. Doumen, T. Lange, and J.-J. Oosterwijk. Faster batch forgery identification. Dec. 2012.

[20] 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*, Jan. 2012.

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

[22] M. Blum. How to prove a theorem so no one else can claim it. In *ICM*, Aug. 1986.

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

[24] 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*, Apr. 2016.

[25] J. Bootle, A. Cerulli, E. Ghadafi, J. Groth, M. Hajiabadi, and S. Jakobsen. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In *ASIACRYPT*, Dec. 2017.

[26] J. Bootle and J. Groth. Efficient batch zero knowledge arguments for low degree polynomials. In *PKC*, Mar. 2018.

[27] X. Boyen and B. Waters. Compact group signatures without random oracles. In *EUROCRYPT*, May 2006.

[28] X. Boyen and B. Waters. Full-domain subgroup hiding and constant-size group signatures. In *PKC*, Apr. 2007.

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

[30] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Efficient range proofs for confidential transactions. In *IEEE S&P*, May 2018.

[31] Cffi. https://bitbucket.org/cffi/cffi.

[32] M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, and G. Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *ACM CCS*, Oct. 2017.

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

[34] A. Chiesa, E. Tromer, and M. Virza. Cluster computing in zero knowledge. In *EUROCRYPT*, Apr. 2015.

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

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

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

[38] Debian, the unversal operating system. https://www.debian.org.

[39] 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 *IEEE S&P*, May 2016.

[40] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography. In *STOC*, 1991.

[41] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, Aug. 1986.

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

[43] D. Fiore, R. Gennaro, and V. Pastro. Efficiently verifiable computation on encrypted data. In *ACM CCS*, Nov. 2014.

[44] M. Fredrikson and B. Livshits. ZØ: An optimizing distributing zero-knowledge compiler. In *USENIX Security*, Aug. 2014.

[45] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *CRYPTO*, Aug. 1997.

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

[47] I. Giacomelli, J. Madsen, and C. Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In *USENIX Security*, Aug. 2016.

[48] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, 1991.

[49] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4):27:1–27:64, Aug. 2015. Preliminary version STOC 2008.

[50] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Computing*, 18(1):186–208, 1989.

[51] J. Groth. Simulation-sound nizk proofs for a practical language and constant size group signatures. In *ASIACRYPT*, Dec. 2006.

[52] J. Groth. Linear algebra with sub-linear zero-knowledge arguments. In

CRYPTO, Aug. 2009.

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

[54] J. Groth and Y. Ishai. Sub-linear zero-knowledge argument for correctness of a shuffle. In *EUROCRYPT*, Apr. 2008.

[55] J. Groth and A. Sahai. Efficient non-interactive proof systems for bilinear groups. In *EUROCRYPT*, Apr. 2008.

[56] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM J. Computing*, 28(4):1364–1396, 1999.

[57] S. Hohenberger, S. Myers, R. Pass, and abhi shelat. ANONIZE: A large-scale anonymous survey system. In *IEEE S&P*, May 2014.

[58] Hyrax reference implementation. `https://github.com/hyraxZK`.

[59] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *IEEE CCC*, June 2007.

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

[61] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, Dec. 2010.

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

[63] J. Kilian. Improved efficient arguments (preliminary version). In *CRYPTO*, pages 311–324, Aug. 1995.

[64] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TRUESET: Faster verifiable set computations. In *USENIX Security*, Aug. 2014.

[65] B. Libert, S. Ramanna, and M. Yung. Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In *ICALP*, July 2016.

[66] libsnark. `https://github.com/scipr-lab/libsnark`.

[67] libSTARK. `https://github.com/elibensasson/libSTARK`.

[68] Y. Lindell. Parallel coin-tossing and constant-round secure two-party computation. *J. Cryptology*, 16(3):143–184, 2003.

[69] H. Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *IACR TCC*, 2011.

[70] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, Oct. 1992.

[71] U. Maurer. Unifying zero-knowledge proofs of knowledge. In *AFRICACRYPT*, June 2009.

[72] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, Aug. 1987.

[73] S. Micali. Computationally sound proofs. *SIAM J. Computing*, 30(4):1253–1298, 2000.

[74] MIRACL crypto SDK. `https://libraries.docs.miracl.com/`.

[75] M. Naor. Bit commitment using pseudorandomness. *J. Cryptology*, 4(2):151–158, 1991.

[76] A. Naveh and E. Tromer. PhotoProof: Cryptographic image authentication for any set of permissible transformations. In *IEEE S&P*, May 2016.

[77] A. V. Oppenheim and A. S. Willsky. *Signals and Systems*. Pearson, 1996.

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

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

[80] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, Aug. 1991.

[81] Pepper project. `https://github.com/pepper-project`.

[82] Reference implementation of the Picnic post-quantum signature scheme. `https://github.com/Microsoft/Picnic`.

[83] N. Pippenger. On the evaluation of powers and monomials. *SIAM J. Computing*, 9(2):230–250, 1980.

[84] A. Poelstra. Updates on confidential transactions efficiency. Sent to the bitcoin-dev email list. `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-December/015346.html`.

[85] D. Pointcheval and J. Stern. Security proofs for signature schemes. In *EUROCRYPT*, May 1996.

[86] PyPy. `https://pypy.org`.

[87] O. Reingold, G. N. Rothblum, and R. D. Rothblum. Constant-round interactive proofs for delegating computation. In *STOC*, June 2016.

[88] SageMath. `http://www.sagemath.org/`.

[89] A. Sahai. Non-malleable non-interactive zero knowledge and adaptive

chosen-ciphertext security. In *FOCS*, Oct. 1999.

[90] C. P. Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991.

[91] 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.

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

[93] 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.

[94] V. Shoup. NTL: A library for doing number theory. `http://www.shoup.net/ntl/`.

[95] E. G. Straus. Addition chains of vectors (problem 5125). *Amer. Math. Monthly*, 70:806–808, 1964.

[96] O. Tange. GNU Parallel: The command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, 2011.

[97] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, Aug. 2013. Full version: `https://arxiv.org/abs/1304.3812`.

[98] J. Thaler. A note on the GKR protocol. `http://people.seas.harvard.edu/~jthaler/GKRNote.pdf`, 2015.

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

[100] K. Turkowski. Filters for common resampling tasks. In *Graphics Gems*, pages 147–165. Academic Press, 1990.

[101] M. Venkitasubramaniam. Personal communication, 2018.

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

[103] R. S. Wahby, M. Howald, S. Garg, abhi shelat, and M. Walfish. Verifiable ASICs. In *IEEE S&P*, May 2016.

[104] R. S. Wahby, Y. Ji, A. J. Blumberg, abhi shelat, J. Thaler, M. Walfish, and T. Wies. Full accounting for verifiable outsourcing. In *ACM CCS*, Oct. 2017. Full version: `https://eprint.iacr.org/2017/242/`.

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

[106] R. S. Wahby, I. Tzialla, abhi shelat, J. Thaler, and M. Walfish. Doubly efficient zkSNARKS without trusted setup. Cryptology ePrint Archive, Report 2017/1132, 2017.

[107] ZCash. `https://z.cash`.

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

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

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

[111] ZKBOO. `https://github.com/Sobuno/ZKBoo`.

A Instantiations of commitment schemes

In this section, we review the Pedersen commitment scheme [80] (Fig. 4) and related protocols.

Theorem 7 ([80]). *The Pedersen commitment scheme is a non-interactive commitment scheme assuming the hardness of the discrete logarithm problem in $\mathcal{G}$.*

Knowledge of opening. Schnorr [90] shows how $\mathcal{P}$ can give a ZK proof that it knows an x, r such that $C_0 = \mathsf{Com}(x; r)$.

Theorem 8 ([90]). proof-of-opening *is complete, honest-verifier perfect ZK, and special sound under the discrete log assumption.*

Commitment to the same value. Using similar ideas, $\mathcal{P}$ can show in ZK that $C_1 = \mathsf{Com}(v_1; s_1)$ and $C_2 = \mathsf{Com}(v_2; s_2)$ are

PEDERSEN COMMITMENT SCHEME

Definitions: Let $\mathcal{G}$ be a (multiplicative) cyclic group of prime order $q_{\mathcal{G}}$ with group operation $\odot$ and inverse $\oslash$. $\mathcal{V}$ publishes generators $g, h \in \mathcal{G}$.

Com(m): $\mathcal{P}$ picks $s \overset{\$}{\leftarrow} \{1, \ldots, q_{\mathcal{G}}\}$ and sends $g^m \odot h^s$.

Open(α): $\mathcal{P}$ sends (m, s). $\mathcal{V}$ checks $\alpha \overset{?}{=} g^m \odot h^s$.

Multi-commitments: For commitments to vectors, $\mathcal{V}$ publishes generators $g_1, \ldots, g_n, h \in G$ and $\mathcal{P}$ sends
$$\text{Com}((m_1, \ldots, m_n)) = h^s \odot \left(\bigodot_i\right) g_i^{m_i}$$

FIGURE 4—The Pedersen commitment scheme.

---

proof-of-product(X, Y, Z)

Inputs: $X = g^x \odot h^{r_X}$, $Y = g^y \odot h^{r_Y}$, and $Z = g^{x \cdot y} \odot h^{r_Z}$. $\mathcal{P}$ knows x, y, r_X, r_Y, and r_Z.

1. $\mathcal{P}$ picks $b_1, \ldots, b_5 \overset{\$}{\leftarrow} \{1, \ldots, q_{\mathcal{G}}\}$ and sends
$$\alpha \leftarrow g^{b_1} \odot h^{b_2} \quad \beta \leftarrow g^{b_3} \odot h^{b_4} \quad \delta \leftarrow X^{b_3} \odot h^{b_5}$$

2. $\mathcal{V}$ sends a challenge $c \overset{\$}{\leftarrow} \{1, \ldots, q_{\mathcal{G}}\}$

3. $\mathcal{P}$ sends
$$z_1 \leftarrow b_1 + c \cdot x \quad z_2 \leftarrow b_2 + c \cdot r_x \quad z_3 \leftarrow b_3 + c \cdot y$$
$$z_4 \leftarrow b_4 + c \cdot r_y \quad z_5 \leftarrow b_5 + c(r_z - r_x y)$$

4. $\mathcal{V}$ checks that
$$\alpha \odot X^c \overset{?}{=} g^{z_1} \odot h^{z_2} \tag{7}$$
$$\beta \odot Y^c \overset{?}{=} g^{z_3} \odot h^{z_4} \tag{8}$$
$$\delta \odot Z^c \overset{?}{=} X^{z_3} \odot h^{z_5} \tag{9}$$

FIGURE 5—ZK proof of knowledge for a product relationship (§A.1).

commitments to the same value, i.e., $v_1 = v_2$. Given $C_u = \text{Com}(u; s_u)$ and a value v, $\mathcal{P}$ can also convince $\mathcal{V}$ that $u = v$.

Theorem 9 (Folklore). proof-of-equality *is complete, honest-verifier perfect zero-knowledge, and special sound under the discrete log assumption.*

A.1 Proving a product relationship

Figure 5 gives a protocol in which $\mathcal{P}$ convinces $\mathcal{V}$ that it has openings to three Pedersen commitments having a product relationship. This is folklore; for example, we know that Maurer [71] describes a very similar protocol.

Theorem 10. *Given commitments X, Y, and Z, proof-of-product proves that Z is a commitment to the product of the values committed in X and Y. This protocol is complete, honest-verifier perfect zero-knowledge, and special sound under the discrete log assumption.*

The proof of Theorem 10 is standard; we leave it to the full version [106] because of space constraints.

---

proof-of-dot-prod($\xi, \tau, \vec{a}$)

Inputs: Commitments $\xi = \text{Com}(\vec{x}; r_\xi)$, $\tau = \text{Com}(y; r_\tau)$, and a vector $\vec{a}$, where $\vec{x}, \vec{a} \in \mathbb{Z}_{q_{\mathcal{G}}}^n$ and $y = \langle \vec{x}, \vec{a} \rangle \in \mathbb{Z}_{q_{\mathcal{G}}}$. $\mathcal{P}$ knows $\vec{x}, r_\xi, y$, and r_τ.

1. $\mathcal{P}$ samples the vector $\vec{d} \overset{\$}{\leftarrow} \{1, \ldots, q_{\mathcal{G}}\}^n$ and the values $r_\beta, r_\delta \overset{\$}{\leftarrow} \{1, \ldots, q_{\mathcal{G}}\}$ and sends
$$\delta \leftarrow \text{Com}(\vec{d}; r_\delta) = h^{r_\delta} \odot \left(\bigodot_i\right) g_i^{d_i} \tag{10}$$
$$\beta \leftarrow \text{Com}(\langle \vec{a}, \vec{d} \rangle; r_\beta) = g^{\langle \vec{a}, \vec{d} \rangle} \odot h^{r_\beta} \tag{11}$$

2. $\mathcal{V}$ sends a challenge $c \overset{\$}{\leftarrow} \{1, \ldots, q_{\mathcal{G}}\}$.

3. $\mathcal{P}$ sends
$$\vec{z} \leftarrow c \cdot \vec{x} + \vec{d}, \quad z_\delta \leftarrow c \cdot r_\xi + r_\delta, \quad z_\beta \leftarrow c \cdot r_\tau + r_\beta$$

4. $\mathcal{V}$ checks that
$$\xi^c \odot \delta \overset{?}{=} \text{Com}(\vec{z}; z_\delta) = h^{z_\delta} \odot \left(\bigodot_i\right) g_i^{z_i} \tag{12}$$
$$\tau^c \odot \beta \overset{?}{=} \text{Com}(\langle \vec{z}, \vec{a} \rangle; z_\beta) = g^{\langle \vec{z}, \vec{a} \rangle} \odot h^{z_\beta} \tag{13}$$

FIGURE 6—ZK vector dot-product proof (§A.2).

A.2 Proving a dot-product relationship

In the protocol of Figure 6, $\mathcal{P}$ convinces $\mathcal{V}$ that it has openings to one multi-commitment $\xi = \text{Com}(\vec{x}; r_\xi)$ and one scalar commitment $\tau = \text{Com}(y; r_\tau)$ such that, for a supplied vector $\vec{a}$ it holds that $y = \langle \vec{x}, \vec{a} \rangle$. Intuitively, this protocol works because
$$\langle \vec{z}, \vec{a} \rangle = \langle c\vec{x} + \vec{d}, \vec{a} \rangle = c\langle \vec{x}, \vec{a} \rangle + \langle \vec{d}, \vec{a} \rangle = cy + \langle \vec{d}, \vec{a} \rangle$$
The above identity is verified in the exponent in Equation (13).

Theorem 11. *The protocol of Figure 6 is complete, honest-verifier perfect zero-knowledge, and special sound under the discrete log assumption.*

The proof of Theorem 11 is standard; we leave it to the full version [106] because of space constraints.

A.3 Dot-product argument from Bulletproofs

The dot-product argument of Appendix A.2 has communication $4+n$ elements for a vector of length n. By adapting the Bulletproof recursive reduction of Bünz et al. [30], we reduce this to $4+2\log n$. Figures 7 and 8 detail this protocol.

As in Appendix A.2, we have $\vec{x}$, $\vec{a}$, and $y = \langle \vec{x}, \vec{a} \rangle$, where $n = |\vec{x}| = |\vec{a}|$. Given $\vec{a}$ and $\Upsilon = \text{Com}_{\vec{g}}(\vec{x}) \odot \text{Com}(y)$, each recursive call to bullet-reduce produces $\vec{a}'$ and $\Upsilon' = \text{Com}_{\vec{g}'}(\vec{x}') \odot \text{Com}(y')$ such that $y' = \langle \vec{x}', \vec{a}' \rangle$.

After $\log n$ such recursive calls, we are left with a scalar $\hat{a}$ and a commitment $\hat{\Upsilon} = \hat{g}^{\hat{x}} g^{\hat{y}} h^{\hat{r}_\Upsilon}$. $\mathcal{P}$ can now use a Schnorr proof to convince $\mathcal{V}$ that $\hat{y} = \hat{x} \cdot \hat{a}$. Expanding Equation (14) (Fig. 8),

$$\left(\hat{\Upsilon}^c \odot \beta\right)^{\hat{a}} \odot \delta = \left(\hat{g}^{c \cdot \hat{x}} \odot g^{c \cdot \hat{y}} \odot h^{c \cdot \hat{r}_\Upsilon} \odot g^d \odot h^{r_\beta}\right)^{\vec{a}} \odot \delta$$
$$= \left(\hat{g}^{c \cdot \hat{x}} \odot g^{c \cdot \hat{y} + d} \odot h^{c \cdot \hat{r}_\Upsilon + r_\beta}\right)^{\vec{a}} \odot \hat{g}^d \odot h^{r_\delta}$$
$$= \hat{g}^{c \cdot \hat{x} \cdot \hat{a} + d} \odot g^{\hat{a}(c \cdot \hat{y} + d)} \odot h^{\hat{a}(c \cdot r_\Upsilon + r_\beta) + r_\delta}$$
$$= \hat{g}^{c \cdot \hat{y} + d} \odot g^{\hat{a}(c \cdot \hat{y} + d)} \odot h^{z_2}$$
$$= \left(\hat{g} \odot g^{\hat{a}}\right)^{z_1} \odot h^{z_2}$$

941

bullet-reduce$(\Upsilon, \vec{a}, \vec{g})$

Inputs: $\Upsilon = h^{r_\Upsilon} \odot g^y \odot \bigodot_{i=1}^{n} g_i^{x_i}$, $\vec{x}, \vec{a} \in \mathbb{Z}_{q_\mathcal{G}}^n$, $y, r_\Upsilon \in \mathbb{Z}_{q_\mathcal{G}}$.

$\mathcal{P}$ knows $\vec{x}, y$, and r_Υ.

Define $\vec{x}_1 = (x_1, \ldots, x_{n/2})$, $\vec{x}_2 = (x_{1+n/2}, \ldots, x_n)$ and similarly for $\vec{a}_1, \vec{a}_2, \vec{g}_1$, and $\vec{g}_2$; and define

$$(g_1, g_2 \ldots)^k \circ (g_{1+n/2}, g_{2+n/2} \ldots)^\ell = (g_1^k \odot g_{1+n/2}^\ell, g_2^k \odot g_{2+n/2}^\ell, \ldots)$$

1. If $n = 1$, return (Υ, a_1, g_1).

2. $\mathcal{P}$ samples $r_{\Upsilon_{-1}}, r_{\Upsilon_1} \xleftarrow{\$} \{1, \ldots, q_\mathcal{G}\}$ and sends

$$\Upsilon_{-1} \leftarrow h^{r_{\Upsilon_{-1}}} \odot g^{\langle \vec{x}_1, \vec{a}_2 \rangle} \odot \bigodot_{i=1}^{n/2} g_{i+n/2}^{x_i}$$

$$\Upsilon_1 \leftarrow h^{r_{\Upsilon_1}} \odot g^{\langle \vec{x}_2, \vec{a}_1 \rangle} \odot \bigodot_{i=1}^{n/2} g_i^{x_{i+n/2}}$$

3. $\mathcal{V}$ chooses and sends $c \xleftarrow{\$} \{1, \ldots, q_\mathcal{G}\}$.

4. $\mathcal{P}$ and $\mathcal{V}$ both compute

$$\Upsilon' \leftarrow \Upsilon_{-1}^{c^2} \odot \Upsilon \odot \Upsilon_1^{c^{-2}}$$

$$\vec{a}' \leftarrow c^{-1} \cdot \vec{a}_1 + c \cdot \vec{a}_2$$

$$\vec{g}' \leftarrow \vec{g}_1^{c^{-1}} \circ \vec{g}_2^{c} = (g_1, \ldots, g_{n/2})^{c^{-1}} \circ (g_{1+n/2}, \ldots, g_n)^c$$

$\mathcal{P}$ computes

$$\vec{x}' \leftarrow c \cdot \vec{x}_1 + c^{-1} \cdot \vec{x}_2$$

$$y' \leftarrow c^2 \cdot \langle \vec{x}_1, \vec{a}_2 \rangle + y + c^{-2} \cdot \langle \vec{x}_2, \vec{a}_1 \rangle$$

$$r_\Upsilon' \leftarrow r_{\Upsilon_{-1}} \cdot c^2 + r_\Upsilon + r_{\Upsilon_1} \cdot c^{-2}$$

5. Return bullet-reduce$(\Upsilon', \vec{a}', \vec{g}')$.

If $y = \langle \vec{x}, \vec{a} \rangle$, then $y' = \langle \vec{x}', \vec{a}' \rangle$, and $\mathcal{P}$ knows $\vec{x}', y', r_\Upsilon'$.

FIGURE 7—Reduction step for the protocol of Figure 8.

---

proof$_{\log}$-of-dot-prod$(\xi, \tau, \vec{a})$

Inputs: $\xi = \mathsf{Com}_{\vec{g}}(\vec{x}; r_\xi) = h^{r_\xi} \odot \bigodot_{i=1}^{n} g_i^{x_i}$,
$\tau = \mathsf{Com}(y; r_\tau) = g^y \odot h^{r_\tau}$. $\vec{x}, \vec{a} \in \mathbb{Z}_{q_\mathcal{G}}^n$, $y, r_\xi, r_\tau \in \mathbb{Z}_{q_\mathcal{G}}$.

$\mathcal{P}$ knows $\vec{x}$, y, r_ξ, and r_τ.

1. Let $\Upsilon = \xi \odot \tau = h^{r_\Upsilon} \odot g^y \odot \bigodot_{i=1}^{n} g_i^{x_i}$ where $r_\Upsilon = r_\tau + r_\xi$. $(\hat{\Upsilon}, \hat{a}, \hat{g}) \leftarrow$ bullet-reduce$(\Upsilon, \vec{a}, \vec{g})$ (see Fig. 7).

At this point, $n = 1$ and $\hat{\Upsilon} = \hat{g}^{\hat{x}} \odot g^{\hat{y}} \odot h^{\hat{r}_\Upsilon}$ where $\hat{y} = \hat{x} \cdot \hat{a}$.

2. $\mathcal{P}$ samples $d, r_\delta, r_\beta \xleftarrow{\$} \{1, \ldots, q_\mathcal{G}\}$ and sends

$$\delta \leftarrow \mathsf{Com}_{\hat{g}}(d; r_\delta) = \hat{g}^d \odot h^{r_\delta}$$

$$\beta \leftarrow \mathsf{Com}_g(d; r_\beta) = g^d \odot h^{r_\beta}$$

3. $\mathcal{V}$ chooses and sends $c \xleftarrow{\$} \{1, \ldots, q_\mathcal{G}\}$.

4. $\mathcal{P}$ sends $z_1 \leftarrow d + c \cdot \hat{y}$ and $z_2 \leftarrow \hat{a} \left(c \cdot \hat{r}_\Upsilon + r_\beta \right) + r_\delta$.

5. $\mathcal{V}$ checks that

$$\left(\hat{\Upsilon}^c \odot \beta \right)^{\hat{a}} \odot \delta \stackrel{?}{=} \left(\hat{g} \odot g^{\hat{a}} \right)^{z_1} \odot h^{z_2} \qquad (14)$$

FIGURE 8—Protocol for dot-product relation based on Bulletproofs [30]. $\mathsf{Com}_{\vec{g}}$ indicates a multi-commitment over generators $\vec{g}$.

In total, $\mathcal{P}$ sends $2 \log n$ elements during the bullet-reduce calls and 4 elements for the final Schnorr proof. Adapting suggestions by Poelstra [84], $\mathcal{V}$'s work computing $\hat{g}$ can be reduced to one multi-exponentiation of length n and one field inversion, and computing $\hat{\Upsilon}$ costs one multi-exponentiation of length $1 + 2 \log n$.

Lemma 12. *The protocol of Figures 7–8 is complete, honest-verifier perfect ZK, and has witness-extended emulation under the discrete log assumption.*

Completeness follows from the derivation of Equation (14) above and the completeness of bullet-reduce [30, Thm. 2, Appx. A], and ZK follows from standard reverse-ordering techniques. Witness-extended emulation follows from the properties of Schnorr protocols and an argument similar to the proof of [30, Thm. 2, Appx. A]. In total, the extractor requires $n + 2$ transcripts.

B Hyrax-I pseudocode

In this section, we provide pseudocode for Hyrax-I. Figure 10 details $\mathcal{V}$'s work; Figures 9 and 11 detail $\mathcal{P}$'s. Our presentation borrows from Wahby et al. [104].

```
1:  function HYRAX-PROVE(ArithCircuit c, input x, witness w, parameter ι)
2:      // Commit to the rows of T via commitments T₁, …, T_{|w|^{1/k}}
3:      SendToVerifier(T₀, …, T_{|w|^{1/k}−1})    // see Line 3 of Figure 10
4:      b_N ← log N, b_G ← log G
5:      (q'₀, q₀,L) ← ReceiveFromVerifier()    // see Line 8 of Figure 10
6:      μ₀,₀ ← 1, μ₀,₁ ← 0, q₀,R ← q₀,L
7:      a₀ ← Com(Ṽ_y(q'₀, q₀,L); 0)
8:      d ← c.depth
9:
10:     for i=1,…,d do
11:         (X, Y, q'ᵢ, qᵢ,L, qᵢ,R) ← ZK-SUMCHECKP(c, i, aᵢ₋₁, μᵢ₋₁,₀, μᵢ₋₁,₁,
12:                                 q'ᵢ₋₁, qᵢ₋₁,L, qᵢ₋₁,R)
13:         if i<d then
14:             (μᵢ,₀, μᵢ,₁) ← ReceiveFromVerifier()    // see Line 21 of Figure 10
15:             aᵢ ← X^{μᵢ,₀} ⊙ Y^{μᵢ,₁}
16:
17:         // Compute Coefficients of the degree b_G polynomial H: H₀, …, H_{log G}
18:         SendToVerifier(Com(H₀), …, Com(H_{b_G}))    // see Line 21 of Figure 10
19:         for i = 0, …, b_G do
20:             proof-of-opening (Com(Hᵢ))
21:         proof-of-equality (Com(H₀), X)
22:         proof-of-equality (Com(H_{b_G}) ⊙ … ⊙ Com(H₀), Y)
23:         τ ← ReceiveFromVerifier()    // see Line 30 of Figure 10
24:         q_d ← (q'_d, (1 − τ) · q_{d,L} + τ · q_{d,R})
25:         ζ = Com(H_{b_G})^{τ^{log G}} ⊙ Com(H_{b_G−1})^{τ^{log G−1}} ⊙ … ⊙ Com(H₀)
26:         T' ← ⊙_{i=0}^{|w|^{1/k}−1} Tᵢ^{χ̌ᵢ}    // χ̂_b is defined in Section 6
27:         R ← (χ̂₀, χ̂_{|w|^{1/ι}}, …, χ̂_{|w|^{1/ι}·(|w|^{1/ι}−1)})    // χ̂_b is defined in Section 6
28:         proof_{log}-of-dot-prod (T'^{q_d[0]}, ζ ⊘ g^{(1−q_d[0])Ṽ_x(q_d[1,…,b_N+b_G−1])}, R)
```

FIGURE 9—Pseudocode for $\mathcal{P}$ in Hyrax-I (§7). The ZK-SumCheckP subroutine is defined in Figure 11. $\mathcal{V}$'s work is described in Figure 10. For notational convenience, we assume $|x| = |w|$, as in Section 6.1.

Left column:

1: **function** HYRAX-VERIFY(ArithCircuit c, input x, output y, parameter ι)

2:     // Receive commitments to the rows of the matrix T

3:     $(T_0, \ldots, T_{|w|^{1/\iota}}) \leftarrow$ ReceiveFromProver()   // see Line 3 of Figure 9

4:     $b_N \leftarrow \log N$, $b_G \leftarrow \log G$

5:     $(q'_0, q_{0,L}) \overset{R}{\leftarrow} \mathbb{F}^{b_N} \times \mathbb{F}^{b_G}$

6:     $\mu_{0,0} \leftarrow 1, \mu_{0,1} \leftarrow 0, q_{0,R} \leftarrow q_{0,L}$

7:     $a_0 \leftarrow \text{Com}(\tilde{V}_y(q'_0, q_{0,0}); 0)$

8:     SendToProver($(q'_0, q_{0,0})$)   // see Line 5 of Figure 9

9:     $d \leftarrow c.\text{depth}$

10:

11:     **for** i=1,...,d **do**

12:         $(X, Y, r', r_L, r_R) \leftarrow$ ZK-SUMCHECKV$(i, a_{i-1}, q'_{i-1}, q_{i-1,L}, q_{i-1,R})$

13:         // $X = \text{Com}(v_0), Y = \text{Com}(v_1)$

14:

15:         **if** i<d **then**

16:             // Pick the next random $\mu_{i,0}, \mu_{i,1}$ and

17:             // compute random linear combination (§3.2)

18:             $\mu_{i,0}, \mu_{i,1} \overset{R}{\leftarrow} \mathbb{F}$

19:             $a_i \leftarrow X^{\mu_{i,0}} \odot Y^{\mu_{i,1}}$

20:             $(q'_i, q_{i,L}, q_{i,R}) \leftarrow (r', r_L, r_R)$

21:             SendToProver($\mu_{i,0}, \mu_{i,1}$)   // see Line 14 of Figure 9

22:

23:     // For the final check, reduce from two points to one point (§3.2)

24:     $(\text{Com}(H_0), \ldots, \text{Com}(H_{b_G})) \leftarrow$ ReceiveFromProver()   // see Line 18 of Figure 9

25:     **for** $i = 0, \ldots, b_G$ **do**

26:         proof-of-opening $(\text{Com}(H_i))$

27:     proof-of-equality $(\text{Com}(H_0), X)$

28:     proof-of-equality $(\text{Com}(H_{b_G}) \odot \ldots \odot \text{Com}(H_0), Y)$

29:     $\tau \overset{R}{\leftarrow} \mathbb{F}$

30:     SendToProver(τ)   // see Line 23 of Figure 9

31:     $q_d \leftarrow (r', (1-\tau) \cdot r_L + \tau \cdot r_R)$

32:     $\zeta = \text{Com}(H_{b_G})^{\tau^{\log G}} \odot \text{Com}(H_{b_G-1})^{\tau^{\log G-1}} \odot \ldots \odot \text{Com}(H_0)$

33:     $T' \leftarrow \bigodot_{i=0}^{|w|^{1/\iota}-1} T_i^{\check{\chi}_i}$ // $\check{\chi}_b$ is defined in Section 6

34:     $R \leftarrow (\hat{\chi}_0, \hat{\chi}_{|w|^{1/\iota}}, \ldots, \hat{\chi}_{|w|^{1/\iota} \cdot (|w|^{1/\iota}-1)})$ // $\hat{\chi}_b$ is defined in Section 6

35:     proof$_{\log}$-of-dot-prod $(T'^{q_d[0]}, \zeta \oslash g^{(1-q_d[0])\tilde{v}_x(q_d[1,\ldots,b_N+b_G-1])}, R)$

36:     **return** accept

37:

38: **function** ZK-SUMCHECKV(layer i, $a_{i-1}, q'_{i-1}, q_{i-1,L}, q_{i-1,R}$)

39:     $(r', r_L, r_R) \overset{R}{\leftarrow} \mathbb{F}^{\log N} \times \mathbb{F}^{\log G} \times \mathbb{F}^{\log G}$

40:     $r \leftarrow (r', r_L, r_R)$

41:     **for** $j = 1, \ldots, \log N + 2\log G$ **do**

42:         $\alpha_j \leftarrow$ ReceiveFromProver()   // α_j is $\text{Com}(s_j)$; see Lines 19,47 of Figure 11

43:         SendToProver($r[j]$)   // see Lines 20,48 of Figure 11

44:     $(X, Y, Z) \leftarrow$ ReceiveFromProver()   // see Line 52 of Figure 11

45:     // $X = \text{Com}(v_0), Y = \text{Com}(v_1), Z = \text{Com}(v_0 v_1)$

46:     // $\mathcal{V}$ computes $\{M_j\}$ as defined in Equation (5)

47:     proof-of-sum-check $(a_{i-1}, \{\alpha_j\}, \{M_j\}, X, Y, Z)$ // see Figure 1

48:     **return** $(\text{Com}(v_0), \text{Com}(v_1), r', r_L, r_R)$

FIGURE 10—Pseudocode for $\mathcal{V}$ in Hyrax-I (§7). $\mathcal{P}$'s work is described in Figures 9 and 11. For notational convenience, we assume $|x| = |w|$, as in Section 6.1.

Right column:

1: **function** ZK-SUMCHECKP(ArithCircuit c, layer i, a_{i-1},)

2:         $\mu_{i-1,0}, \mu_{i-1,1}, q'_{i-1}, q_{i-1,L}, q_{i-1,R}$

3:     **for** $j = 1, \ldots, b_N$ **do**

4:         // In these rounds, prover sends commitment to degree-3 polynomial s_j

5:         **for all** $\sigma \in \{0,1\}^{b_N - j}$ and $g \in \{0,1\}^{b_G}$ and $k \in \{-1, 0, 1, 2\}$ **do**

6:             $s \leftarrow (g, g_L, g_R)$ // g_L, g_R are labels of g's layer-i inputs in sub-circuit.

7:             termP $\leftarrow \widetilde{eq}(q'_{i-1}, r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b_N - j]) \cdot$

8:                 $(\mu_{i-1,0} \cdot \chi_g(q_{i-1,L}) + \mu_{i-1,1} \cdot \chi_g(q_{i-1,R}))$

9:             termL $\leftarrow \tilde{V}_i(r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b_N - j], g_L)$

10:             termR $\leftarrow \tilde{V}_i(r'[1], \ldots, r'[j-1], k, \sigma[1], \ldots, \sigma[b_N - j], g_R)$

11:

12:             **if** g is an add gate **then** $s_j[\sigma, g][k] \leftarrow$ termP $\cdot$ (termL + termR)

13:             **else if** g is a mult gate **then** $s_j[\sigma, g][k] \leftarrow$ termP $\cdot$ termL $\cdot$ termR

14:

15:         **for** $k \in \{-1, 0, 1, 2\}$ **do**

16:             $s_j[k] \leftarrow \sum_{\sigma \in \{0,1\}^{b_N - j}} \sum_{g \in \{0,1\}^{b_G}} s_j[\sigma, g][k]$

17:

18:         // Compute coefficients of s_j and create a multi-commitment (§5)

19:         SendToVerifier($\text{Com}(s_j)$)   // see Line 42 of Figure 10

20:         $r'[j] \leftarrow$ ReceiveFromVerifier()   // see Line 43 of Figure 10

21:

22:     $r' \leftarrow (r'[1], \ldots, r'[b_N])$ // notation

23:

24:     **for** $j = 1, \ldots, 2b_G$ **do**

25:         // In these rounds, prover sends commitment to degree-2 polynomial $s_{b_N + j}$.

26:         **for all** gates $g \in \{0,1\}^{b_G}$ and $k \in \{-1, 0, 1\}$ **do**

27:             $s \leftarrow (g, g_L, g_R)$ // g_L, g_R are labels of g's layer-i inputs in subcircuit

28:             $u_{k,0} \leftarrow (q_{i-1,L}[1], \ldots, q_{i-1,L}[b_G], r[1], \ldots, r[j-1], k)$

29:             $u_{k,1} \leftarrow (q_{i-1,R}[1], \ldots, q_{i-1,R}[b_G], r[1], \ldots, r[j-1], k)$

30:             termP $\leftarrow \widetilde{eq}(q'_{i-1}, r') \cdot (\mu_{i,0} \cdot \prod_{\ell=1}^{b_G+j} \chi_{s[\ell]}(u_{k,0}[\ell]) +$

31:                 $\mu_{i,1} \cdot \prod_{\ell=1}^{b_G+j} \chi_{s[\ell]}(u_{k,1}[\ell]))$

32:

33:             **if** $j \leq b_G$ **then**

34:                 termL $\leftarrow \tilde{V}_i(r', r[1], \ldots, r[j-1], k, g_L[j+1], \ldots, g_L[b_G])$

35:                 termR $\leftarrow \tilde{V}_i(r', g_R)$

36:             **else**   // $b_G < j \leq 2b_G$

37:                 termL $\leftarrow \tilde{V}_i(r', r[1], \ldots, r[b_G])$

38:                 termR $\leftarrow \tilde{V}_i(r', r[b_G+1], \ldots, r[j-1], k, g_R[j-b_G+1], \ldots, g_R[b_G])$

39:

40:             **if** g is an add gate **then** $s_{b_N+j}[g][k] \leftarrow$ termP $\cdot$ (termL + termR)

41:             **else if** g is a mult gate **then** $s_{b_N+j}[g][k] \leftarrow$ termP $\cdot$ termL $\cdot$ termR

42:

43:         **for** $k \in \{-1, 0, 1\}$ **do**

44:             $s_{b_N+j}[k] \leftarrow \sum_{g \in \{0,1\}^{b_G}} s_{b_N+j}[g][k]$

45:

46:         // Compute coefficients of s_{b_N+j} and create a multi-commitment (§5)

47:         SendToVerifier($\text{Com}(s_{b_N+j})$)   // see Line 42 of Figure 10

48:         $r[j] \leftarrow$ ReceiveFromVerifier()   // see Line 43 of Figure 10

49:

50:     $r_0 \leftarrow (r[1], \ldots, r[b_G])$ $r_1 \leftarrow (r[b_G + 1], \ldots, r[2b_G])$ // notation

51:     $v_0 \leftarrow \tilde{V}_i(r', r_0)$ $v_1 \leftarrow \tilde{V}_i(r', r_1)$ // $X = \text{Com}(v_0), Y = \text{Com}(v_1), Z = \text{Com}(v_0 v_1)$

52:     SendToVerifier(X, Y, Z)   // see Line 44 of Figure 10

53:     // $\mathcal{P}$ computes $\{M_k\}$ as defined in Equation (5).

54:     proof-of-sum-check $(a_{i-1}, \{\text{Com}(s_j)\}, \{M_k\}, X, Y, Z)$ // see Figure 1

55:

56:     **return** $(\text{Com}(v_0), \text{Com}(v_1), r', r_0, r_1)$

FIGURE 11—$\mathcal{P}$'s side of the zero-knowledge sum-check protocol in Hyrax-I (§7).