GARUDA and PARI: Faster and Smaller SNARKs via Equifficient Polynomial Commitments

Michel Dellepere
mdelle1@alumni.stanford.edu
Independent

Pratyush Mishra
prat@upenn.edu
UPenn

Alireza Shirzad
alrshir@upenn.edu
UPenn

Abstract

SNARKs are powerful cryptographic primitives that allow a prover to produce a succinct proof of a computation. Two key goals of SNARK research are to minimize the size of the proof and to minimize the time required to generate it. In this work, we present new SNARK constructions that push the frontier on both of these goals.

Our first construction, PARI, is a SNARK that achieves the smallest proof size amongst *all* known SNARKs. Specifically, PARI achieves a proof size of just two group elements and two field elements, which, when instantiated with the BLS12-381 curve, totals just 160 bytes. This is smaller than the sizes for Groth16 [Groth, EUROCRYPT '16] and Polymath [Lipmaa, CRYPTO '24]. PARI also achieves the lowest known gas cost for on-chain SNARK verification, reducing the gas cost by 6% compared to Groth16 and 17% compared to FFLONK.

Our second construction, GARUDA, is a SNARK that reduces proof generation time by supporting, for the first time, arbitrary "custom" gates and *free* linear gates (in terms of cryptographic costs) for non-uniform computations. These benefits enable significant prover-time savings compared to state-of-the-art SNARKs.

Both constructions rely on a new cryptographic primitive: "equifficient" polynomial commitment (EPC) schemes that enforce that committed polynomials have the same representation in particular bases. We provide both rigorous security definitions for this primitive as well as efficient constructions for univariate and multilinear polynomials.

Our constructions are obtained via a new compiler that obtains a succinct argument by combining polynomial IOPs with our EPC schemes.

1 Introduction

Succinct Non-interactive ARguments of Knowledge (SNARKs) are cryptographic proofs that enable a prover to efficiently convince a computationally weak verifier of claims of the form "Given a program P and public input $\mathbb{x}$, I know a private input $\mathbb{w}$ such that $P(\mathbb{x},\mathbb{w}) = 1$". A *zero-knowledge (zk)*-SNARK additionally ensures that the proof reveals no information about the witness beyond the validity of the claim.

(zk-)SNARKs have found numerous applications in practice, including in blockchain systems [5, 41, 51], verifiable machine learning [42, 46], and verifiable image transformations [22], privacy-preserving network packet inspection [53], and more. Two fundamental metrics in the study of SNARKs are *prover time* and *proof size*.

Prover time. From a prover time perspective, all known SNARKs require time at least linear in the size of the circuit C being proven, and so much effort has been invested in reducing the size of C. An important line of work in this direction has been the study of more expressive circuits that support "custom" gates that can compute arbitrary functions of their inputs, rather than just addition and multiplication. Examples of such gates include high-degree polynomial gates [26] and lookup gates [28]. State-of-the-art SNARKs are able to prove such expressive circuits in time linear in the circuit size [17, 49]. Unfortunately, the proving algorithms of these SNARKs perform cryptographic operations that scale with the number of linear *and* non-linear gates in the circuit. In contrast, prior SNARKs for simpler circuit models [30, 32] only pay cryptographic costs for non-linear gates, and are able to prove linear gates "for free". A key open question in the literature has been whether we can obtain the best of both worlds and construct SNARKs for non-uniform computations that support both custom gates *and* free linear gates.

This question is interesting from both theoretical and practical perspectives. From a theory standpoint, cheap linear gates would align custom-gate SNARKs not only with prior SNARKs [30, 32], but also with other cryptographic primitives such as MPC [21, 38] and FHE [31] where linear operations are often much cheaper than non-linear ones. From a practical perspective, linear gates can comprise a significant fraction of circuits that arise in practice, and so eliminating cryptographic costs for them can lead to significant performance improvements. Indeed, we show in Section 8 that constraint systems that support free linear gates can be much

smaller than those that do not.

Proof size. From the proof-size perspective, the state-of-the-art SNARKs are Groth16 [32] and Polymath [39]. Over the BLS12-381 curve, Groth16 proofs are just 192 bytes, while Polymath proofs are 176 bytes. An important open question has thus been: what is the shortest proof size for a SNARK for NP?

1.1 Our contributions

We answer all the foregoing questions in the affirmative by constructing two new SNARKs: GARUDA and PARI. While both schemes share a common construction methodology, they exhibit different performance profiles and target distinct application scenarios. We detail the ideas behind these SNARKs next.

GARUDA: custom gates and free linear gates. GARUDA is a SNARK for *Generalized* Rank-1 Constraint Satisfiability (GR1CS), an NP-complete language that we introduce, which extends the popular Rank-1 Constraint Satisfiability (R1CS) with support for custom gates. For a constraint system of size n, GARUDA's prover requires $O(n)$ field operations and $O(n)$ group operations. It also achieves $O(\log n)$ proof size and verifier time. We prove (an interactive version of) GARUDA secure in the algebraic group model (AGM) [25].

PARI: a 2-group element SNARK. PARI is a new SNARK for 'square' R1CS [34] that achieves a proof size of two group elements (and two field elements) over Type-III pairing-friendly groups. When instantiated with the BLS12-381 curve, PARI achieves a proof size of just 1280 bits, which is the smallest in the literature. More generally, no matter the choice of pairing-friendly curve, PARI's proof size is always smaller than that of the state-of-the-art prior work [32, 39]. The prover and verifier times for PARI are also similar to those for Groth16 [32]: for a circuit of size n, PARI's prover requires $O(n \log n)$ field operations and $O(n)$ group operations, while its verifier cost is dominated by 3 pairings. Like GARUDA, we prove an interactive version of PARI secure in the AGM.

New methodology: succinct arguments from equifficient polynomial commitments. We present a methodology for constructing succinct arguments for GR1CS that extends the popular 'Polynomial Interactive Oracle Proof (PIOP) + Polynomial Commitment (PC)' framework [19, 14] to work with a new kind of PC scheme that enforces "equal-coefficient", or *equifficient*, constraints. Roughly, such equifficient PC (EPC) schemes enforce the additional property that given a batch of polynomials $p_1, \ldots, p_n$ and associated bases $\mathscr{B}_1, \ldots, \mathscr{B}_n$, the coefficient vectors of these polynomials in their respective bases are equal.

Our new 'PIOP + *EPC*' methodology leverages EPC schemes to enforce linear constraints, as opposed to prior PIOP-based constructions [19, 48] which use complex PIOPs for this task. This shift enables constructions of succinct arguments that can use simpler PIOPs that are responsible only for non-linear checks.

We provide a generic construction of EPC schemes from any pairing-based (plain) PC schemes where the commitment to a polynomial is a Pedersen-like commitment [45]. A number of popular PC schemes satisfy this constraint, including the KZG [36] and PST [44] schemes.

We also describe an extension that allow producing a single *batch commitment* to multiple polynomials where the commitment size is independent of the number of polynomials in the batch. This property is crucial for attaining the small proof size in our construction of PARI. Our constructions are inspired by prior "Linear PCP"-based SNARKs [35, 9]. We believe that our EPC schemes could find application in places where PC schemes are used today (e.g., verifiable secret sharing); we leave these explorations to future work.

Implementation and evaluation. We implement GARUDA and PARI in a new library built atop the `arkworks` framework [20]. Our library flexibly extends the constraint-writing framework of `arkworks` to support GR1CS. We use our implementation to compare the performance of both SNARKs to numerous baselines, and show that when benchmarked on the same computation (iterations of the Rescue-Prime [50] hash function), our implementation of GARUDA is almost 5× faster than Groth16 and 2.67× faster than HyperPlonk, thus demonstrating that the combination of free linear gates and custom gates indeed leads to much faster proving times. We also benchmark GARUDA on real-world application circuits of interest, and show that even without custom gates, it offers improved proving times over Groth16. We also show that PARI achieves comparable native verification cost and better verification cost in blockchain applications than Groth16. Concretely, we implement a Solidity verifier for PARI and demonstrate that it achieves the lowest known gas cost among state-of-the-art SNARKs for on-chain verification, reducing gas consumption by 6% compared to Groth16 and 17% compared to FFLONK [27].

Remark 1.1 (circuit-specific setup)**.** GARUDA *and* PARI *require a* circuit-specific *trusted setup. While not ideal, we view this as a worthwhile trade-off for the benefits (such as faster on-chain verification) we gain in return. Indeed, numerous blockchain applications today rely on the circuit-specific Groth16 [32] zkSNARK for this reason. It is also likely that for both SNARKs, one can design trusted-setup ceremonies that closely resemble existing ceremonies for Groth16 [13].*

2 Technical overview

To understand the techniques behind GARUDA and PARI, it is instructive to first recall how prior SNARKs for R1CS are constructed, and why they fall short of our goals. We focus on R1CS both because we will later generalize it to add support for custom gates, and because it is the constraint system of choice for many SNARKs [30, 8, 32, 19, 48] and underlies

popular circuit programming frameworks [4, 20].

Background: R1CS. Recall that the NP relation $\mathcal{R}_{\mathsf{R1CS}}$ is the set of triples $(\mathbb{i},\mathbb{x},\mathbb{w}) = ((\mathbb{F},m,A,B,C),x,w)$ where $\mathbb{F}$ is a finite field, A,B,C are matrices in $\mathbb{F}^{m\times m}$, and $z := (x,w) \in \mathbb{F}^m$ satisfies $Az \circ Bz = Cz$.

In R1CS, addition gates are captured via the linear combinations introduced by the matrix-vector products Az, Bz, Cz, while multiplication gates are captured by the Hadamard product relation between the latter. Thus, a SNARK with "free" addition gates for R1CS would pay cryptographic prover costs that scale only with the cost of the Hadamard product, as opposed to costs that scale also with those of the matrix-vector multiplications. In other words, the cryptographic work of such a SNARK scales with the number of rows in the matrix, as opposed to the number of non-zero entries in the matrix.

2.1 Circuit-specific-setup SNARKs for R1CS

The checks performed by all existing SNARKs for R1CS can be decomposed into two complementary kinds:

- *Linear checks (linchecks)* enforce that there exist vectors z, z_A, z_B, z_C satisfying $z_A = Az$, $z_B = Bz$, and $z_C = Cz$, and
- *Non-linear checks (rowchecks)* enforce that $z_A \circ z_B = z_C$.

Existing circuit-specific SNARKs perform these checks as follows.

For **linear checks**, these SNARKs rely on the following probabilistic test for each matrix $M \in \{A,B,C\}$: $\langle \boldsymbol{r}, z_M \rangle \stackrel{?}{=} \langle \boldsymbol{r}^\top M, z \rangle$, where $\boldsymbol{r} \leftarrow \mathbb{F}^m$ is a random vector. In fact, for efficiency, $\boldsymbol{r}$ is replaced by $\mathcal{L}^K(\tau)$, which is a vector whose i-th element is the i-th Lagrange polynomial $\mathcal{L}_i^K(X)$ for $K \subset \mathbb{F}$, evaluated at a random field element $\tau \leftarrow \mathbb{F} \setminus K$. ($K$ has size equal to the number of constraints m). The check thus becomes $\langle \mathcal{L}^K(\tau), z_M \rangle \stackrel{?}{=} \langle \mathcal{L}^K(\tau)^\top M, z \rangle$, and its soundness follows from the Schwartz–Zippel lemma [47, 54].

To construct a SNARK from this check, the key insight in prior work [30] is that the special form of the vector $\mathcal{L}^K(X)$ allows us to write the foregoing check as $\langle \mathcal{L}^K(\tau), z_M \rangle = \langle \boldsymbol{m}(\tau), z \rangle$, where $\boldsymbol{m}(X)$ is the vector whose i-th element $m_i(X) = \langle \mathcal{L}^K(X), M_i \rangle$ is the polynomial interpolating the i-th column of M. For efficiency, these checks can then be further batched together via random coefficients $\alpha_A, \alpha_B, \alpha_C$:

$$\textstyle\sum_{M\in\{A,B,C\}} \alpha_M \cdot \langle \mathcal{L}^K(\tau), z_M \rangle = \langle \sum_{M\in\{A,B,C\}} \alpha_M \cdot \boldsymbol{m}(\tau), z \rangle$$

To compile this into a SNARK, existing works [30, 8] rely on a pairing-friendly group $\langle \mathsf{group} \rangle = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, G, H, e)$ as follows. A setup phase, on input the R1CS index $(\mathbb{F}, m, A, B, C)$, samples $\alpha_A, \alpha_B, \alpha_C \leftarrow \mathbb{F}$ and $\tau \leftarrow \mathbb{F} \setminus K$, and constructs the proving key $\mathsf{pk} = (\boldsymbol{\Sigma}_1 = \mathcal{L}^K(\tau) \cdot G, \boldsymbol{\Sigma}_2 = (\sum_M \alpha_M \boldsymbol{m}(\tau)) \cdot G)$ and the verification key $\mathsf{vk} = (\alpha_A H, \alpha_B H, \alpha_C H)$. The prover, for each $M \in \{A,B,C\}$, commits to z_M via the Pedersen [45] commitment $c_M := \langle z_M, \boldsymbol{\Sigma}_1 \rangle$ and to z via the Pedersen commitment $c := \langle z, \boldsymbol{\Sigma}_2 \rangle$. The verifier checks that the linear relation is satisfied via the following pairing check:

$$\prod_{M\in\{A,B,C\}} e(c_M, \alpha_M H) = e(c, H) \quad . \tag{1}$$

This SNARK is highly efficient, requiring only 4 m-sized $\mathbb{G}_1$-MSMs from the prover, and 4 pairings from the verifier. It can be proven sound in the Algebraic Group Model [25].

For **non-linear row-wise checks**, SNARKs for R1CS rely on the following polynomial identity:

$$\hat{z}_A(X) \cdot \hat{z}_B(X) - \hat{z}_C(X) = 0 \mod v_K(X) \quad . \tag{2}$$

Here $\hat{z}_M(X) = \langle \mathcal{L}^K(X), z_M \rangle$ is the polynomial interpolating z_M, and $v_K(X) = \prod_{k\in K}(X-k)$ is the polynomial that is 0 at every point in K. (For an exposition of this identity, see prior work [30, 7]). To see how we can compile this check into a SNARK, notice that the group elements c_A, c_B, c_C in the linear-check SNARK are Pedersen commitments to $\hat{z}_A, \hat{z}_B, \hat{z}_C$ respectively. Prior work [30, 8] performs the check directly over these commitments via pairings:

First, the prover's commitment c_B to $\hat{z}_B$ is changed to be over $\mathbb{G}_2$ instead of $\mathbb{G}_1$. Second, the prover provides a commitment $c_h \in \mathbb{G}_1$ to the polynomial $h(X) := (\hat{z}_A(X) \cdot \hat{z}_B(X) - \hat{z}_C(X))/v_K(X)$. Then, the verifier uses the following pairing equation to check Equation (2) at a random point τ in the exponent:

$$e(c_A, c_B) = e(c_C, H) \cdot e(c_h, v_K(\tau) \cdot H) \tag{3}$$

This check can be proven sound in the AGM [25].

Shortcomings. This approach has shortcomings for both goals: small proof size and support for custom gates.

On the proof size front, even optimized versions of this approach, namely Groth16 [32] require the proof to include a witness-dependent $\mathbb{G}_2$ element for the non-linear check, and it seems like this is inherent for publicly verifiable SNARKs. Because $\mathbb{G}_2$ elements are larger than $\mathbb{G}_1$ elements, this results in a proof size that is larger than necessary. This also worsens verifier time as one cannot take advantage of preprocessing techniques to speed up pairings involving this witness-dependent $\mathbb{G}_2$ element.

On the prover time front, this approach cannot generalize to support higher-degree custom gates, e.g. checks of the form $Az \circ Bz \circ Cz = Dz$. This is because non-linear checks rely on bilinear maps to multiply polynomials "in the exponent", and the latter only supports degree-2 multiplication.

2.2 Universal-setup SNARKs for R1CS

Notice that the shortcomings of the prior approach stem from limitations imposed by the non-linear rowchecks. Recent work on *universal-setup* SNARKs bypasses these limitations via a new approach that directly checks identities like those of Eq. (2) *in plain*, outside the exponent.

These SNARKs follow the popular 'PIOP + PC scheme $\to$ SNARK' methodology [19, 14]. Briefly, this methodology combines two components: a Polynomial Interactive Oracle Proof (PIOP), and a Polynomial Commitment (PC) scheme. We describe these components in detail in Section 4, but briefly, a PIOP is an interactive proof system where the prover's messages are polynomials, and the verifier does not read these messages but instead queries them at evaluation points of its choice, while a PC scheme is a cryptographic tool that allows the prover to commit to a polynomial and later prove that it evaluates to a claimed value at a claimed point. One can combine these components to construct a SNARK by replacing the PIOP prover's polynomials with commitments to them, and then using the PC scheme to prove that the commitments are consistent with the PIOP verifier's queries.

Shortcomings. Unfortunately, in existing constructions that follow this approach [19, 15, 48, 49], the PIOP is responsible for both linear *and* non-linear checks. While non-linear checks like those of Eq. (2) have efficient PIOPs, linear checks require PIOPs whose proving costs scale with the number of non-zero entries in the matrix, and which require the prover to compute numerous oracles. After compilation to a SNARK, this results in proving costs that require cryptographic work for addition gates, and also result in larger proof sizes.

2.3 Our approach

To overcome these issues, we propose a new method to combine the best of both worlds: we use efficient PIOPs for non-linear custom gates, efficient linear-check SNARKs for linear gates. To do so, we introduce a new notion of *equifficient* PCs that allows us to link the two components cleanly.

We will use as a running example the following generalization of R1CS that enforces that $Az \circ Bz \circ Cz = Dz$ for some matrices A, B, C, D. This generalization can be seen as enforcing a degree-3 custom gate.

- Linchecks: We can adapt the linear-check SNARK from Section 2.1 for our generalized R1CS by simply increasing the number of matrices in the check of Eq. (1): $\prod_{M \in \{A,B,C,D\}} e(c_M, \alpha_M H) = e(c, H)$.
- Rowchecks: Instead of proving the R1CS identity of Eq. (2) via pairings directly, we can use a PIOP that enforces a generalization of Eq. (2), i.e., $\hat{z}_A(X) \cdot \hat{z}_B(X) \cdot \hat{z}_C - \hat{z}_D(X) = 0 \mod v_K(X)$, by sending the additional quotient polynomial $h(X) := (\hat{z}_A(X) \cdot \hat{z}_B(X) \cdot \hat{z}_C - \hat{z}_D(X))/v_K(X)$. One can compile this PIOP into a SNARK for rowcheck via, e.g., the KZG PC scheme [36]; the resulting SNARK requires commitments to $\hat{z}_A(X)$, $\hat{z}_B(X)$, $\hat{z}_C$, $\hat{z}_D(X)$, and h, and an opening proof for the evaluation of these at a random point.

The key thing that is left to do now is to link the commitments created in rowcheck with those created in lincheck. But this is actually trivial: we can simply use the same commitments for both! In particular, the commitment c_M is actually a *KZG* commitment to the polynomial $\hat{z}_M(X)$.

In the rest of this paper, we formalize this high-level intuition by introducing a new type of PC scheme that allows us to provide not only to commit to polynomials and then later provide evaluation proofs for them, but also to enforce that the coefficients of the committed polynomials satisfy particular linear constraints. In particular, our tool, called *equifficient* polynomial commitment (EPC) schemes, enforce a particular constraint: that the coefficients of a list of polynomials in a given basis are equal. We elaborate on this next in Section 3, and show how to formalize the foregoing informal construction in Section 4.

3 Equifficient polynomial commitments

We recall the standard notion of PC schemes [36], and then describe our new notion of *equifficient* PC that extends the standard one. Throughout, fix a finite field $\mathbb{F}$ and a polynomial vector space $\mathbb{K}$ over $\mathbb{F}$ consisting of polynomials of size D.[1]

3.1 Background: polynomial commitments

Formally, a PC scheme for polynomials of size D is a tuple of algorithms $\mathsf{PC} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ with the following syntax.

$\mathsf{PC.Setup}(1^\lambda, D)$ samples public parameters $(\mathsf{ck}, \mathsf{ok}, \mathsf{vk})$. $\mathsf{PC.Commit}(\mathsf{ck}, p)$ produces a commitment c to a polynomial $p \in \mathbb{K}$. One can prove that the committed polynomial evaluates to v at a point u by producing an evaluation proof π using $\mathsf{PC.Open}(\mathsf{ok}, p, u)$. Finally, the verifier can check the evaluation claim by running $\mathsf{PC.Verify}(\mathsf{vk}, c, u, v, \pi)$, which accepts if and only if the proof confirms that c corresponds to a polynomial that evaluates to v at u.

For use in SNARKs, PC schemes must be *complete* and *extractable* [19]. Roughly, the latter property says that whenever the verifier is convinced by an adversarial committer's evaluation proof, then the committer must "know" a polynomial underlying the commitment that evaluates to the claimed value at the claimed point. We defer the formal definition to Section B.4.

3.2 Definition of EPC schemes

An *equifficient polynomial commitment scheme* (EPC) is a PC scheme in the standard sense with the additional property that equifficient constraints are enforced on committed polynomials. We define these constraints next.

3.2.1 Coefficient-equality constraints

We extend the standard notion of extractability by enforcing a new property called *coefficient equality*. Let $p_1, \ldots, p_n$ be polynomials in $\mathbb{K}$. Associate with each p_i a basis $\mathscr{B}_i$ of $\mathbb{K}$, and denote by $[p_i]_{\mathscr{B}_i}$ the coefficients of p_i when expressed

[1] For example, univariate polynomials of degree $D-1$ or multilinear polynomials in $\log D$ variables.

in the basis $\mathscr{B}_i$. Then, a coefficient-equality (or *equifficient*) constraint Λ on polynomials $\boldsymbol{p} = [p_1, \ldots, p_n]$ in $\mathbb{K}$ is a list $[\mathscr{B}_1, \ldots, \mathscr{B}_n]$ of bases for $\mathbb{K}$ that enforces that the coefficient vectors of the polynomials in their respective bases are equal; that is, $[p_1]_{\mathscr{B}_1} = \cdots = [p_n]_{\mathscr{B}_n}$. We denote this constraint satisfaction by the predicate $\Lambda(\boldsymbol{p}) = 1$. If Λ is specified by a single basis, then we call the constraint *trivial*, and assume without loss of generality the basis is the canonical basis for $\mathbb{K}$, denoted by $\mathscr{U}$.[2]

3.2.2 Definition

An EPC scheme consists of algorithms whose syntax and properties are as follows.

Syntax. EPC.Setup samples public parameters pp containing a description of $\mathbb{K}$ along with its canonical basis $\mathscr{U}$. The algorithm EPC.Specialize then specializes these public parameters pp for a set of equifficient constraints $\boldsymbol{\Omega} = \{\Omega_1, \ldots, \Omega_s\}$, constructing committer, opener, and verifier keys $(\mathsf{ck}, \mathsf{ok}, \mathsf{vk})$ that collectively enforce the constraints in $\boldsymbol{\Omega}$ on committed polynomials.

The committer can then use EPC.Commit to commit to a list of polynomials $\boldsymbol{p} = [p_1, \ldots, p_n]$ while enforcing that these polynomials are subject to an equifficient constraint $\Lambda \in \boldsymbol{\Omega}$. Later on, the committer can use EPC.Open to produce a proof that the committed polynomials evaluate to claimed evaluations $\boldsymbol{v} = [v_1, \ldots, v_n]$ at a claimed point u. Finally, the verifier can use EPC.Verify to check this proof, with the guarantee that if the proof passes, then the resulting evaluations are correct, and moreover that the committed polynomials satisfy the equifficient constraint Λ.

Extractability. EPC schemes are required to satisfy *equifficient* extractability guarantees. Roughly, this means that, given an equifficient constraint $\Lambda \in \boldsymbol{\Omega}$, for every adversary who can produce a commitment-proof pair that causes EPC.Verify to accept, there exists an efficient extractor that outputs the polynomials in the adversarial commitment that satisfy the evaluation claims *and* the claimed equifficient constraints.

Hiding. We can also optionally require that EPC schemes are hiding, meaning that the commitment c and evaluation proof π do not reveal any information about the committed polynomials $\boldsymbol{p}$ beyond their evaluations at u. We will use this property in Section 4 to ensure that the SNARKs we construct from EPC schemes are zero-knowledge.

We note that formalizing the foregoing informal description in a way that suffices for our application requires some care. For instance, we generalize our definition to support committing to batches of polynomials such that each batch is subject to a different equifficient constraint. We also consider a strong extractability definition that supports multiple commitments produced across multiple rounds of interaction. We formalize these in the full version.

[2] For example, for univariate polynomials of degree at most D, the canonical basis is $\mathscr{U} = \{1, X, X^2, \ldots, X^{D-1}\}$.

3.3 Construction from Pedersen-like PC schemes

We construct an equifficient PC scheme (EPC) from any Pedersen-like polynomial commitment scheme PC and a zk-SNARK for linear subspaces Π_{LS}. Throughout, fix a bilinear group $\langle \mathsf{group} \rangle = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, G, H, e)$ of prime order q with generators G, H.

3.3.1 Building block: Linear subspace SNARKs

To enforce equifficient constraints, we will use a linear subspace SNARK for the relation $\mathcal{R}_{\mathsf{LS}}$ that checks whether a list of group elements $\boldsymbol{c} = [c_1, \ldots, c_t] \in \mathbb{G}_1^t$ can be expressed as the matrix-vector product $M \cdot \boldsymbol{p}$, where $M \in \mathbb{G}_1^{t \times D}$ is a matrix of group elements, and $\boldsymbol{p} \in \mathbb{F}_q^D$ is a vector of coefficients.

3.3.2 Building block: Pedersen-like PC schemes

Let PC be a PC scheme where the commitment key ck contains a set of group elements $\boldsymbol{\Sigma} = [G_1, \ldots, G_D] \in \mathbb{F}_q^D$. The commitment algorithm PC.Commit takes as input a polynomial p whose coefficients in the canonical basis $\mathscr{U}$ are $[p_1, \ldots, p_D]$, and outputs a commitment $c := \sum_{i=1}^{D} p_i \cdot G_i$

3.3.3 Construction

$\mathsf{EPC.Setup}(1^\lambda, D)$ invokes PC.Setup to obtain keys $(\mathsf{ck}_{\mathsf{PC}}, \mathsf{vk}_{\mathsf{PC}})$ and sets $\mathsf{pp} = \mathsf{ck}$.

To specialize to a single equifficient constraint $\Omega = [\mathscr{B}_1, \ldots, \mathscr{B}_t]$, where each basis $\mathscr{B}_i$ is a list of linearly-independent polynomials $[b_j^{(i)}]_{j=1}^{D}$ (expressed in the canonical basis), the algorithm EPC.Specialize proceeds as follows:

- Encode each basis $\mathscr{B}_i$ by committing to basis elements: $B_i := [\mathsf{PC.Commit}(\mathsf{ck}_{\mathsf{PC}}, b_1^{(i)}), \ldots, \mathsf{PC.Commit}(\mathsf{ck}_{\mathsf{PC}}, b_D^{(i)})]$.
- Construct a matrix M whose i-th row is the vector of group elements B_i.
- Run the generator for the subspace SNARK with respect to M: $(\mathsf{ipk}_{\mathsf{LS}}, \mathsf{ivk}_{\mathsf{LS}}) \leftarrow \mathcal{G}_{\mathsf{LS}}(1^\lambda, M)$.
- Output committer key $\mathsf{ck} := (\mathsf{ck}_{\mathsf{PC}}, \mathsf{ipk}_{\mathsf{LS}})$, opener key $\mathsf{ok} := \mathsf{ok}_{\mathsf{PC}}$, and verification key $\mathsf{vk} := (\mathsf{vk}_{\mathsf{PC}}, \mathsf{ivk}_{\mathsf{LS}})$.

Commit. To commit to polynomials $p_1, \ldots, p_t \in \mathbb{K}$ whose coefficient vectors in the bases $\mathscr{B}_1, \ldots, \mathscr{B}_t$ are equal, the committer converts each p_j to its representation p_j' in the canonical basis. It then computes the commitments $c_j = \mathsf{PC.Commit}(\mathsf{ck}, p_j') \in \mathbb{G}_1$ for each $j \in [t]$. Finally, it proves that the polynomials $p_1, \ldots, p_t$ have the same coefficient vector when expressed in the bases $\mathscr{B}_1, \ldots, \mathscr{B}_t$ by constructing a SNARK proof $\pi_{\mathsf{LS}} \leftarrow \mathcal{P}_{\mathsf{LS}}(\mathsf{ipk}_{\mathsf{LS}}, (c_1, \ldots, c_t), p)$, where p is the shared coefficient vector. The final commitment is $c = [c_1, \ldots, c_t, \pi_{\mathsf{LS}}]$.

Open. To open the commitment at a point $x \in \mathbb{F}_q$, the opener computes evaluation proofs $\pi_j = \mathsf{PC.Open}(\mathsf{ck}, p_j', x)$ for each $j \in [t]$. The full proof is $\pi = (\pi_1, \ldots, \pi_t)$.

Verify. To verify a proof $\pi = (\pi_1, \ldots, \pi_t)$ that claims that polynomials committed in $c = [c_1, \ldots, c_t, \pi_{\mathsf{LS}}]$ evaluate to $[y_1, \ldots, y_t]$ at a point x, the verifier first checks that each evaluation proof π_j is valid by checking that $\mathsf{PC.Verify}(\mathsf{vk}, c_j, x, y_j, \pi_j) = 1$ for each $j \in [t]$. Then, it checks that the commitments satisfy the equifficient constraint by checking that $\mathcal{V}'_{\mathsf{LS}}(\mathsf{ivk}_{\mathsf{LS}}, (c_1, \ldots, c_t), \pi_{\mathsf{LS}}) = 1$

3.3.4 Extractability

At a high level, by extractability of the underlying PC scheme PC, if the verifier accepts an evaluation proof, then one can extract polynomials $p_1, \ldots, p_t$ that evaluate to the claimed values $y_1, \ldots, y_t$ at the claimed point x. Hence, we are left to show that these polynomials indeed have the same coefficient vector when expressed in the bases $\mathscr{B}_1, \ldots, \mathscr{B}_t$. This follows from the soundness of the linear subspace SNARK: if the EPC verifier accepts, then the LS verifier also accepts, which in turn means that there exists a vector p such that $c_i = M_i \cdot p$ for each $i \in [t]$, where M_i is the i-th row of the matrix M constructed in EPC.Specialize.

Now, assume for contradiction that there exists an index $i^\star$ such that the above holds, but the polynomial $p_{i^\star}$ underlying $c_{i^\star}$ has a coefficient vector p' in $\mathscr{B}_{i^\star}$ that does not equal p. That is, there exists a vector $p' \neq p$ such that $p_{i^\star} = \sum_{j=1}^{D} p'_j \cdot b_j^{(i^\star)}$. This would in turn mean that $c_{i^\star} = \mathsf{PC.Commit}(\mathsf{ck}, p_{i^\star}) = \sum_{j=1}^{D} p'_j \cdot \mathsf{PC.Commit}(\mathsf{ck}, b_j^{(i^\star)}) = \sum_{j=1}^{D} p'_j \cdot M_{i^\star, j}$. However, since $\mathscr{B}_{i^\star}$ is a basis, the elements $\mathsf{PC.Commit}(\mathsf{ck}, b_j^{(i^\star)})$ are linearly independent, and hence each p'_j must be equal to the coefficients p_j; otherwise, one can break the binding of the PC scheme.

Remark 3.1. *The foregoing analysis assumes that each $\mathscr{B}_i$ is a basis for $\mathbb{K}$, i.e. that M is full rank. However, in our applications to constructing SNARKs, it might occur that $\mathscr{B}_i$ does not form a basis, or even a set of linearly independent polynomials. In this case, one has to settle for enforcing equifficiency only on those locations in $\mathscr{B}_i$ where the polynomials are indeed linearly independent. This suffices for our applications.*

3.3.5 Extensions and optimizations

In our SNARK constructions, we consider several optimizations and extensions of the foregoing EPC construction.

Batch opening proofs. The construction in Section 3.3.3 provides a single opening proof for each polynomial committed in the EPC commitment. However, many Pedersen-like PC schemes, such as KZG [36], support batch opening proofs, where the prover can produce a single proof that proves the evaluation of multiple polynomials at the same point. Our EPC construction inherits this feature out-of-the-box.

Hiding. Achieving hiding in EPC schemes is helpful for constructing zero-knowledge SNARKs down-the-line. We briefly sketch how to adapt the foregoing EPC construction to achieve hiding. Hiding in Pedersen-based PC schemes is typically achieved by switching to hiding Pedersen commitments where the commitment key contains additional elements $\mathbf{\Sigma}' = [G'_1, \ldots, G'_D]$ that are used to incorporate a random "masking" polynomial into the commitment. The opening proof additionally contains the evaluation of this masking polynomial at the evaluation point, and asserts that this evaluation is consistent with the commitment.

To extend our EPC construction to achieve hiding, we can have EPC.Specialize add to ck additional powers $[\alpha_1 \mathbf{\Sigma}', \ldots, \alpha_t \mathbf{\Sigma}']$ for random $\alpha_i \in \mathbb{F}$. The linear-subspace SNARK would now be defined with respect to the matrix

$$M' = M \,||\, \begin{bmatrix} \alpha_1 \cdot \mathbf{\Sigma}' & 0 & \ldots & 0 \\ 0 & & \ddots & 0 \\ 0 & 0 & \ldots & \alpha_t \cdot \mathbf{\Sigma}' \end{bmatrix},$$

and the commitments and opening proofs for p_i would be computed with respect to the elements $[\mathbf{\Sigma}, \alpha_i \cdot \mathbf{\Sigma}']$.

The astute reader might notice that M' is now no longer full rank, and hence Remark 3.1 now applies. But this is not a concern, as equifficiency is only "violated" for the masking polynomials, and not for the actual polynomials p_i that we care about. Note that this means of achieving hiding is incompatible with the batch opening proofs described above.

4 Succinct arguments from EPC schemes

With this new tool in hand, we are now ready to describe how to construct succinct arguments that meet our goals. We begin by defining a generalized version of R1CS; the SNARK we construct will support this relation. As we will see, our generalization can be specialized both for the small proof case (i.e., Square R1CS) and for the custom gate case.

Generalized R1CS. GR1CS generalizes R1CS by allowing the NP relation to enforce an arbitrary predicate L on vectors $M_1 z, \ldots, M_t z$ for some matrices $M_1, \ldots, M_t$. More precisely,

Definition 4.1 (informal version of Definition B.3)**.** *The NP relation GR1CS is the set of triples $(\mathbb{i}, \mathbb{x}, \mathbb{w}) = ((\mathbb{F}, m, \mathscr{C}), x, w)$ where $\mathbb{F}$ is a finite field and $\mathscr{C}$ is a set of constraints, each of which is a tuple $(M_1, \ldots, M_t, L)$ such that $z := (x, w) \in \mathbb{F}^m$ satisfies $L(M_1 z, \ldots, M_t z) = 0$.*

We now describe how to use an EPC to construct succinct arguments for GR1CS. We make two simplifying assumptions when discussing the techniques.

- We assume that there is a single local predicate. We explain the case for multiple predicates in Section A.
- We assume that the NP instance $\mathbb{x}$ is empty, and defer describing the modifications required to handle non-empty instances to Remark 4.3.

At a high level, our construction follows the 'PIOP + PC' recipe [14, 19, 29], but replaces the standard PC scheme used in that recipe with an *equifficient* PC scheme. This EPC

scheme is then used to enforce linear constraints, while the PIOP is tasked only with enforcing non-linear constraints.

Construction intuition. Recall that the construction attempt in Section 2.3 enforces linear constraints via the randomized check $\sum_{i=1}^{t} \alpha_i \cdot \langle \mathcal{L}^K(\tau), z_{M_i} \rangle \stackrel{?}{=} \langle \sum_{i=1}^{t} \alpha_i \cdot \boldsymbol{m}_i(\tau), z \rangle$. Our insight is that this check can be viewed as an *equifficient constraint* on the polynomials $\hat{z}_{M_1}, \ldots, \hat{z}_{M_t}$: they are required to have the same coefficient representation (i.e., z) in their respective "matrix" bases $(\boldsymbol{m}_1, \ldots, \boldsymbol{m}_t)$. Therefore, to enforce these constraints, we can just require the argument prover to commit to the polynomials $\hat{z}_{M_1}, \ldots, \hat{z}_{M_t}$ with an EPC scheme under the equifficient constraint $\Lambda = (\boldsymbol{m}_1, \ldots, \boldsymbol{m}_t)$. We can then use a PIOP that enforces non-linear constraints over these committed polynomials to complete the argument.

We provide a detailed overview of the construction below.

Construction overview. We begin by introducing some notation. A GR1CS index $\mathbb{i} = (\mathbb{F}, m, \mathscr{C})$ is satisfied by an instance-witness pair $(\mathbb{x}, \mathbb{w}) = (x, w)$ if the local predicate L is satisfied by the vectors $M_1 z, \ldots, M_t z$. In the exposition below, we assume that the matrices $M_1, \ldots, M_t$ are square matrices of size m, and moreover that their columns are linearly-independent (and hence form a basis for $\mathbb{F}^m$).[3] Below $\mathcal{D}$ will be a subset of the field $\mathbb{F}$ of size m. (The particular choice of subset depends on whether we are using univariate or multilinear polynomials; it does not affect the exposition.)

Our construction will rely on PIOPs for the non-linear component of GR1CS, which we characterize as follows:

Definition 4.2. *The rowcheck relation is a tuple* $(\mathbb{i}, \mathbb{x}, \mathbb{w}) = ((\mathbb{F}, L, \mathcal{D}), [[\![\hat{z}_i]\!]]_{i=1}^{t}, [\hat{z}_i]_{i=1}^{t})$*, where the index consists of a field* $\mathbb{F}$*, a local predicate* L*, and a subset* $\mathcal{D}$ *of* $\mathbb{F}$*. The instance contains polynomial oracles* $[[\![\hat{z}_1]\!]], \ldots, [[\![\hat{z}_t]\!]]$*, while the witness contains the corresponding polynomials. A triple is in the relation if the local predicate is satisfied by the polynomials at all points in* $\mathcal{D}$*, i.e., for all* $x \in \mathcal{D}$*,* $L(\hat{z}_1(x), \ldots, \hat{z}_t(x)) = 0$*.*

Given a PIOP for rowcheck, and an appropriate compatible equifficient PC scheme EPC, we are now ready to describe our SNARK construction.

Generator. On input a GR1CS index $\mathbb{i} = (\mathbb{F}, m, ([M_i]_{i=1}^{t}, L))$, the argument generator $\mathcal{G}$ samples proving and verification keys $(\mathsf{ipk}, \mathsf{ivk})$ as follows.

First, $\mathcal{G}$ samples EPC public parameters via $\mathsf{EPC.Setup}$. Then, for each matrix M_i, $\mathcal{G}$ constructs the basis set $\mathscr{M}_i := [\hat{m}_{i,j}]_{j=1}^{k}$ where $\hat{m}_{i,j}$ is a polynomial that interpolates the j-th column of M_i over the domain $\mathcal{D}$. It then invokes the EPC specialization algorithm with input the equifficient constraint $\Lambda := (\mathscr{M}_1, \ldots, \mathscr{M}_t)$ to obtain the commitment keys $(\mathsf{ck}, \mathsf{ok}, \mathsf{vk})$ specialized to these constraints. (It also ensures that these keys allow committing to *unconstrained* polynomials.) We note that it is this specialization step that samples randomness specific to the GR1CS instance, and hence leads to a circuit-specific proving key.

Next, it constructs from the GR1CS index $\mathbb{i}$ a corresponding rowcheck index, and invokes the PIOP indexer on the latter to obtain indexer polynomials $\boldsymbol{p}_0$. It commits to these via $\mathsf{EPC.Commit}$ to obtain the commitment c_0.

Finally, $\mathcal{G}$ constructs the proving key $\mathsf{ipk} = (\mathsf{ck}, \mathsf{ok}, \boldsymbol{p}_0)$ and the verification key $\mathsf{ivk} = (\mathsf{vk}, c_0)$, and outputs these.

Prover and verifier. We describe the interaction between the argument prover $\mathcal{P}$ and verifier $\mathcal{V}$. The prover gets as input the proving key ipk, the GR1CS instance x, and the GR1CS witness w, while the verifier gets as input the verification key ivk and the instance x. (Recall that in this high-level exposition, we assume that the GR1CS instance x is empty. We include it here syntactically anyway for familiarity.)

The prover $\mathcal{P}$ starts by setting $z := (x, w)$, and then computing the vectors $z_i := M_i \cdot z \in \mathbb{F}^m$ for each i in $1, \ldots, t$. It then interpolates these vectors over the domain $\mathcal{D}$ to get the polynomials $\hat{z}_1, \ldots, \hat{z}_t$. $\mathcal{P}$ uses $\mathsf{EPC.Commit}$ to commit to these polynomials under the equifficient constraint Λ, and sends the resulting commitments to the verifier $\mathcal{V}$.

The prover and verifier then proceed as in the methodology of Chiesa et al. [19], i.e., by simulating the rowcheck PIOP prover and verifier respectively. In each round, instead of sending the polynomials produced by the PIOP prover in the plain, $\mathcal{P}$ instead commits to these using $\mathsf{EPC.Commit}$, but without enforcing any equifficient constraints. At the end of the interaction, when the PIOP verifier wishes to query the committed polynomials, $\mathcal{V}$ sends the query set to $\mathcal{P}$, who responds with the claimed evaluations and the evaluation proofs produced by $\mathsf{EPC.Open}$. $\mathcal{V}$ concludes the protocol by checking that these evaluation proofs are valid, and by checking that PIOP verifier accepts with the provided evaluations.

To compile this to a *non-interactive* argument in the random oracle model, we can invoke the Fiat–Shamir transform [24].

Remark 4.3 (handing non-empty public inputs)**.** *So far we have assumed that the GR1CS instance* x *is empty. When* x *is non-empty, the GR1CS variable assignment* z *is the concatenation* $(x \| w)$*. In our full construction, we handle this by enabling the verifier to directly and soundly obtain oracle access to the correct* $\hat{z}_i$*. The key idea is to force the prover to leave "empty slots" in the commitments to these polynomials that can be filled in with the instance by the verifier.*

In more detail, we leverage the linearity of the low-degree extension operation to write $\hat{z}_i$ *as the sum of* $\hat{x}_i$ *and* $\hat{w}_i$*, where* $\hat{x}_i$ *and* $\hat{w}_i$ *are the LDEs of* $M_i \cdot (x \| 0)$ *and* $M_i \cdot (0 \| w)$*.*

Then, if the prover can be forced to send commitments to $\hat{w}_i$*, the verifier can evaluate* $\hat{z}_i$ *at any point* u *by computing the sum* $\hat{x}_i(u) + \hat{w}_i(u)$*: the first part the verifier can compute itself,*[4] *while the second part will be provided by the prover (along with a corresponding evaluation proof).*

[3] Ensuring that the matrices are square and full-rank can be done by adding an appropriate number of dummy constraints and/or variables.

[4] In general this might cost more $|x|$ field operations, but one can (and we do) apply a transformation from prior work [39] to avoid this blow-up.

To enable this sketch to work, we modify the EPC to support equifficient constraints over "punctured bases" where equifficient constraints are enforced only over certain basis elements, while the remaining "punctured" coefficients are zero. For a detailed discussion of puncturing, see Section B.1.

5 Constructing GARUDA

To construct GARUDA, we instantiate the blueprint from Section 4 with a new multilinear EPC scheme and a PIOP for rowcheck that supports multiple predicates.

Multilinear PIOP for rowcheck. We choose multilinear zero-check PIOP, to check if at all points in $\mathcal{D}$, i.e., for all $x \in \mathcal{D}$, $L(\hat{z}_1(x), \ldots, \hat{z}_t(x)) = 0$. Multilinear zero-check PIOP is based on the multilinear-sumcheck protocol, and allows us to obtain the desired linear prover time.

Multilinear EPC scheme. We instantiate the EPC scheme from Section 3.3.3 with the PST multilinear PC scheme [44] and the linear subspace SNARK from [37] that is proven secure in the AGM [16].

Using these ingredients gives us great efficiency: a commitment to t ν-variate polynomials is of size $t+1$ $\mathbb{G}_1$ elements (t commitments and one linear subspace SNARK proof, each of size 1 $\mathbb{G}_1$ element), while the opening proof, after applying batch opening optimizations, is just ν $\mathbb{G}_1$ elements.

Theorem 1 (informal)**.** GARUDA *(Fig. 1) is a succinct argument for GR1CS that achieves linear cryptographic prover costs and logarithmic proof size and verifier time.*

Zero-knowledge. To achieve zero-knowledge,

- Similar to prior work [29] we avoid the use of hiding Pedersen commitments for $\hat{z}_i$, and instead directly add randomness to $\hat{z}_i$ by adding dummy random constraints.
- To ensure that the opening proofs do not leak information, we require the prover to provide a hiding commitment to a zero polynomial. This polynomial is then batch-opened with the other commitments.
- We apply sumcheck masking techniques used in prior work [18, 52] to prevent leakage from sumcheck messages.

6 Constructing PARI

To construct PARI, we instantiate the blueprint from Section 4 with a new univariate EPC scheme and a univariate PIOP for rowcheck specialized to 'Square' R1CS. We briefly describe this PIOP below, and provide details about the EPC scheme construction in Section 6.1.

Square R1CS [34] is a special case of GR1CS previously introduced by Groth and Maller [34]. It enforces constraints of the form $Az \circ Az = Bz$. This translates to a rowcheck relation where the local predicate is the claim that $\hat{z}_A^2(X) - \hat{z}_B(X) = 0$ on $\mathcal{D}$. A PIOP for this relation is obtained straightforwardly by adapting PIOPs for R1CS 'rowcheck' (i.e., $\hat{z}_A(X) \cdot \hat{z}_B(X) - \hat{z}_C(X) = 0$) from prior work e.g. Aurora [6] or Marlin [19].

Applying the blueprint from Section 4 with the univariate EPC scheme and this univariate PIOP gives the following:

Theorem 2 (informal)**.** PARI *(Fig. 2) is a SNARK for NP with proof size consisting of* 2 $\mathbb{G}_1$ *and* 2 $\mathbb{F}$ *elements.*

6.1 An EPC scheme for univariate polynomials

We construct an EPC scheme for univariate polynomials by instantiating our construction from Section 3.3.3 as follows. For the PC scheme, we use the Pedersen-like KZG PC scheme [36], while for the linear subspace SNARK, we use the construction of Kiltz and Wee [37] that is proven secure by Campanelli et al. in the AGM [16].

These ingredients provide almost ideal efficiency: commitments and opening proofs for KZG are of size 1 $\mathbb{G}_1$ element each, while the linear subspace SNARK also has a proof size of 1 $\mathbb{G}_1$ element, and the verifier has to perform $t+1$ pairing checks to enforce an equifficient constraint on t polynomials. However, this is insufficient for achieving Theorem 2. We further use a new *batch commitment* that allows us to commit to, and open, multiple polynomials with commitment and proof size equal to that required for a *single* polynomial.

To see this, we need to recall the construction of the LS SNARK of Kiltz and Wee [37]. Briefly, to enforce that, given a matrix $M \in \mathbb{G}_1^{t \times D}$ of group elements and a vector $c \in \mathbb{G}_1^t$ of group elements, there exists a scalar vector $p \in \mathbb{F}^D$ such that $c = M \cdot p$, the generator samples $\alpha_1, \ldots, \alpha_t \in \mathbb{F}$ and constructs the row vector $m' = [\alpha_1, \ldots, \alpha_t]^\top \cdot M$. It also samples a random field element a, and publishes the proving key $\mathsf{ipk}_{\mathsf{LS}} = m'$ and the verification key $\mathsf{ivk}_{\mathsf{LS}} = (a \cdot H, [a \cdot \alpha_i H]_{i=1}^t)$. The proof then is $\pi_{\mathsf{LS}} := \langle m', p \rangle = \sum_{i=1}^t \alpha_i \cdot \langle p_i, M_i \rangle$.

Our key observation is that when M is full rank, m' functions as a "batch" commitment key for committing to p, and π_{LS} is the commitment to p (and hence to the polynomials $p_1, \ldots, p_t$). Hence, instead of committing to each polynomial p_i separately, we can commit to them all just once via π_{LS}.

Adjusting the opening proofs can be done as follows. Recall that the KZG opening proof for a polynomial p_i at a point u is a commitment to the polynomial $w_i(X) := (p_i(X) - p_i(u))/(X-u)$. To avoid separate commitments to each w_i, we instead commit to them all at once as $\sum_{i+1}^t \alpha_i \cdot w_i(X)$. To enable this commitment, we publish commitment keys of the form $\mathsf{ck}_i = \alpha_i / \delta \cdot \mathbf{\Sigma}$ for each $i = 1, \ldots, t$. Here δ is a random field element that serves as "domain-separator". The batch opening proof is then computed as $\sum_{i=1}^t \langle w_i, \mathsf{ck}_i \rangle$, and the verifier checks that this is consistent with the commitment π_{LS}.

7 Implementation

We implemented GARUDA and PARI as a Rust library in around 5000 lines of code.[5] Our library is built atop the

[5] https://anonymous.4open.science/r/garuda-pari/

Generator $\mathcal{G}(1^\lambda, \mathbb{i}) \to (\mathsf{ipk}, \mathsf{ivk})$:

1. Parse the index $\mathbb{i}$ as $\big(\mathbb{F}, n, k, m, c, t, [L_i, (M_{i,1}, \ldots, M_{i,t}), m_i]_{i=1}^c\big)$, and set $\nu := \lceil \log m \rceil$.
2. Construct the 'succinct' index $\mathbb{i}_s := (\mathbb{F}, n, k, m, c, t, [L_i, (M'_{i,1}, \ldots, M'_{i,t}), m_i]_{i=1}^c)$ where $M'_{i,j}$ only has the first n columns of $M_{i,j}$.
3. Sample a bilinear group $\langle \mathsf{group} \rangle = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, G, H, e) \leftarrow \mathsf{SampleGrp}(1^\lambda)$.
4. Sample uniformly random field elements $\boldsymbol{\tau} \xleftarrow{\$} \mathbb{F}^\nu$ and $\alpha_1, \ldots, \alpha_t \leftarrow \mathbb{F}$. Set $\mathsf{ivk}_\alpha := (\alpha_1 H, \ldots, \alpha_t H)$.
5. Construct the unconstrained commitment key: $\mathsf{ck}_l := [\chi_i(\boldsymbol{\tau})]_{i=1}^m$.
6. For each $i \in [t]$, construct the stacked matrix M_i^*, and extend its columns over $\mathcal{B}_\nu$ to obtain basis polynomials $\mathscr{M}_i^* = [\hat{m}_{i,j}^*(\mathbf{X})]_{j \in [k]}$.
7. Construct the consistency commitment key $\mathsf{ck}_c := [(\sum_{i=1}^t \alpha_i \hat{m}_{i,j}^*(\boldsymbol{\tau}))G]_{j=n+1}^k$.
8. Construct the consistency verification key $\mathsf{vk}_c := [(\sum_{i=1}^t \alpha_i \hat{m}_{i,j}^*(\boldsymbol{\tau}))G]_{j=1}^n$.
9. Compute the selector polynomials $\mathsf{ipk}_s := [S_i(\mathbf{X})]_{i=1}^c$ as in the PIOP, and commit to them: $\mathsf{ivk}_s := [S_i(\boldsymbol{\tau})G]_{i=1}^c$.
10. Set $\mathsf{ipk} = (\mathbb{i}, \langle \mathsf{group} \rangle, \mathsf{ck}_c, \mathsf{ck}_l, \mathsf{ipk}_s)$ and $\mathsf{ivk} = (\mathbb{i}_s, \langle \mathsf{group} \rangle, \mathsf{ivk}_\alpha, \mathsf{ivk}_s, \mathsf{vk}_c)$, and output the keys $(\mathsf{ipk}, \mathsf{ivk})$.

Prover $\mathcal{P}^\rho(\mathsf{ipk}, \mathbb{x}, \mathbb{w}) \to \pi$:

1. Parse ipk as $(\mathbb{i}, \langle \mathsf{group} \rangle, \mathsf{ck}_c, \mathsf{ck}_l, \mathsf{ipk}_s)$, the instance $\mathbb{x}$ as x, the witness $\mathbb{w}$ as w, and define $z := (x \| w)$.
2. Parse the GR1CS index as $\mathbb{i} = \big(\mathbb{F}, n, k, m, c, t, [L_i, (M_{i,1}, \ldots, M_{i,t}), m_i]_{i=1}^c\big)$.
3. For each $i \in [t]$,
   (a) Construct the vectors $z_i := M_i^* z$ and $w_i := M_i^*(0 \| w)$, and their MLEs over $\mathcal{B}_\nu$: $\hat{z}_i(\mathbf{X})$ and $\hat{w}_i(\mathbf{X})$.
   (b) Commit to $\hat{w}_i(\mathbf{X})$: $c_{w,i} := \hat{w}_i(\boldsymbol{\tau}) \cdot G$.
4. Compute the consistency commitment $c_c := \sum_{i=1}^{k-n} w[i] \cdot \mathsf{ck}_c[i]$ and assemble the commitments $c_w := [c_{w,i}]_{i=1}^t$.
5. Simulate the prover-verifier interaction for the PIOP for the polynomial $P := \sum_{i=1}^c S_i \cdot L_i(\hat{z}_1, \ldots, \hat{z}_t)$. Namely, in the i-th round of the PIOP where $i \in [\nu]$:
   (a) Compute the next prover message and state: $(p_i, \mathsf{st}_{i+1}) \leftarrow \mathbf{P}_i(\mathsf{st}_i, [\boldsymbol{\rho}_j]_{j=1}^{i-1})$.
   (b) Sample a random field element from the random oracle $\rho_i := \rho(\mathsf{transcript})$.
6. Form the set of PIOP prover messages $\pi_p := [p_i]_{i=1}^\nu$.
7. Compute the evaluations $\boldsymbol{v} := [\hat{w}_i(\boldsymbol{\rho})]_{i=1}^t \cup [S_i(\boldsymbol{\rho})]_{i=1}^c$.
8. Sample linear combination coefficients $\zeta_1, \ldots, \zeta_{t+c} := \rho(\mathsf{transcript})$, and use these to obtain the linearly combined polynomial $B(\mathbf{X}) := \sum_{i=1}^c \zeta_i S_i(\mathbf{X}) + \sum_{i=0}^t \zeta_{i+c} \hat{w}_i(\mathbf{X})$.
9. Compute the PST opening proof for B: $\pi_o := [W_i(\boldsymbol{\tau})G]_{i \in [\nu]}$, where the polynomials W_i are given by the equation $B(\mathbf{X}) - B(\boldsymbol{\rho}) = \sum_{i \in [\nu]} (\mathbf{X}[i] - \boldsymbol{\rho}[i]) \cdot W_i(\mathbf{X})$.
10. Output the proof $\pi := (c_w, c_c, \pi_p, \boldsymbol{v}, \pi_o)$

Verifier $\mathcal{V}^\rho(\mathsf{ivk}, \mathbb{x}, \pi) \to \{0, 1\}$:

1. Parse ivk as $(\mathbb{i}_s, \langle \mathsf{group} \rangle, \mathsf{ivk}_\alpha, \mathsf{ivk}_s, \mathsf{vk}_c)$, and $\mathbb{i}_s$ as $(\mathbb{F}, n, k, m, c, t, [L_i, (M'_{i,1}, \ldots, M'_{i,t}), m_i]_{i=1}^c)$.
2. Parse the instance $\mathbb{x}$ as x and the proof π as $(c_w, c_c, \pi_p, \boldsymbol{v}, \pi_o)$.
3. Simulate the prover-verifier interaction for the PIOP for the polynomial $P := \sum_{i=1}^c S_i \cdot L_i(\hat{z}_1, \ldots, \hat{z}_t)$. Namely, in the i-th PIOP round where $i \in [\nu]$:
   (a) Obtain the prover's next message from π_p, and add it to the transcript.
   (b) Sample a random field element from the random oracle: $\rho_i := \rho(\mathsf{transcript})$.
4. For each $i \in [t]$: compute the vector $x_i := M_i^* x$ and the evaluation of its MLE: $\hat{z}_i(\boldsymbol{\rho}) := \hat{w}_i(\boldsymbol{\rho}) + \hat{x}_{M_i}(\boldsymbol{\rho})$.
5. For the last round of the PIOP, provide $\boldsymbol{v}$ and $[\hat{z}_i(\boldsymbol{\rho})]_{i=1}^t$ to the PIOP verifier $\mathbf{V}$ and check the PIOP decision.
6. Check that the equifficient constraint is satisfied: $e(c_c, H) \stackrel{?}{=} \prod_{i=1}^t e(c_w[i], \mathsf{ivk}_\alpha[i])$
7. Sample coefficients $\zeta_1, \ldots, \zeta_{t+c} := \rho(\mathsf{transcript})$, and use these to obtain a commitment c_B to $B(\mathbf{X})$, and the evaluation v_B of B at $\boldsymbol{\rho}$ by linearly combining the commitments to, and evaluations of, S_i and $\hat{w}_i$.
8. Check the evaluation proof for B: $e(c_B - v_B G, H) \stackrel{?}{=} \prod_{i=1}^\nu e(\pi_o[i], \tau_i H - \boldsymbol{\rho}[i] \cdot H)$

Figure 1: The unrolled GARUDA SNARK.

$\mathcal{G}(1^\lambda, \mathbb{i}) \to (\mathsf{ipk}, \mathsf{ivk})$:

1. Parse the Square R1CS index $\mathbb{i}$ as $(\mathbb{F}, n, k, m, A, B)$ where $A \in \mathbb{F}^{m \times k}$, $B \in \mathbb{F}^{m \times k}$.
2. Set the succinct index $\mathbb{i}_s := (\mathbb{F}, n, k, m, A', B')$ where $A' \in \mathbb{F}^{m \times n}$, $B' \in \mathbb{F}^{m \times n}$.
3. Sample a bilinear group $\langle \mathsf{group} \rangle = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, G, H, e) \leftarrow \mathsf{SampleGrp}(1^\lambda)$.
4. Sample random field elements $\alpha, \beta, \delta_1, \delta_2, \tau \leftarrow \mathbb{F}$.
5. Interpolate the columns of A and B over K to obtain the bases $\mathscr{A} := [a_i(X)]_{i=1}^k$ and $\mathscr{B} := [b_i(X)]_{i=1}^k$.
6. Compute the commitment keys $\Sigma = [\frac{\alpha a_i(\tau) + \beta b_i(\tau)}{\delta_2} G]_{i=n+1}^k$ and $\Sigma_q = [\frac{\tau^i}{\delta_2} G]_{i=1}^k$
7. Compute the opening keys: $\Sigma_A := [\frac{\alpha \tau^i}{\delta_1} G]_{i=1}^k$, $\Sigma_B := [\frac{\beta \tau^i}{\delta_1} G]_{i=1}^k$, and $\Sigma'_q := [\frac{\tau^i}{\delta_1} G]_{i=1}^k$.
8. Output $\mathsf{ipk} = (\langle \mathsf{group} \rangle, \mathbb{i}, \Sigma, \Sigma_q, \Sigma_A, \Sigma_B, \Sigma'_q)$ and $\mathsf{ivk} = (\langle \mathsf{group} \rangle, \mathbb{i}_s, \alpha G, \beta G, \delta_1 H, \delta_2 H, \tau H, \delta_1 \tau H)$.

$\mathcal{P}^\rho(\mathsf{ipk}, \mathbb{x}, \mathbb{w}) \to \pi$:

1. Parse ipk as $(\langle \mathsf{group} \rangle, \mathbb{i}, \Sigma, \Sigma_q, \Sigma_A, \Sigma_B, \Sigma'_q)$.
2. Parse the Square R1CS index $\mathbb{i}$ as $(\mathbb{F}, n, k, m, A, B)$ where A, B are matrices in $\mathbb{F}^{m \times k}$.
3. Construct the variable assignment z by concatenating the instance and witness assignments: $z := (x \| w)$.
4. Compute $z_A := Az$ and $z_B := Bz$, and interpolate these over K to obtain polynomials $\hat{z}_A$ and $\hat{z}_B$.
5. Compute $w_A := A \cdot (0 \| w)$ and $w_B := B \cdot (0 \| w)$, and interpolate these over K to obtain polynomials $\hat{w}_A, \hat{w}_B$.
6. Compute a batched commitment to the polynomials $\hat{w}_A$ and $\hat{w}_B$: $T_{AB} := \sum_{i=n+1}^k w_i \cdot \Sigma[i]$.
7. Compute the quotient polynomial $q := \frac{\hat{z}_A^2 - \hat{z}_B}{v_K}$, and commit to it: $T_Q := \sum_{i \in [m]} q[i] \Sigma_q[i]$.
8. Batch the commitments: $T := -(T_{AB} + T_Q)$
9. Sample an evaluation point from the random oracle: $r := \rho(\mathsf{transcript})$.
10. Evaluate $\hat{w}_A$ and $\hat{w}_B$ at r to get evaluations $v_a := \hat{w}_A(r)$ and $v_b := \hat{w}_B(r)$, respectively.
11. Compute the opening witness polynomials:
$$\hat{W}_A(X) := \frac{\hat{w}_A(X) - \hat{w}_A(r)}{(X - r)}; \quad \hat{W}_B(X) := \frac{\hat{w}_B(X) - \hat{w}_B(r)}{(X - r)}; \quad \hat{W}_Q(X) := \frac{q(X) - q(r)}{(X - r)} \quad .$$
12. Compute the separate opening proofs:
$$W_A := \sum_{i \in [m]} \hat{W}_A[i] \cdot \Sigma_A[i]; \quad W_B := \sum_{i \in [m]} \hat{W}_B[i] \cdot \Sigma_B[i]; \quad W_Q := \sum_{i \in [m]} \hat{W}_Q[i] \cdot \Sigma'_q[i]$$
13. Batch the opening proofs: $U := W_A + W_B + W_Q$
14. Output proof $\pi := (T, U, v_a, v_b)$.

$\mathcal{V}^\rho(\mathsf{ivk}, \mathbb{x}, \pi) \to b$:

1. Parse the verification key $\mathsf{ivk} = (\langle \mathsf{group} \rangle, \mathbb{i}_s, \alpha G, \beta G, \delta_1 H, \delta_2 H, \tau H, \delta_1 \tau H)$.
2. Parse the proof π as (T, U, v_a, v_b).
3. Sample an evaluation point from the random oracle $r := \rho(\mathsf{transcript})$.
4. Compute the vectors $x_A := A \cdot (x \| 0)$ and $x_B := B \cdot (x \| 0)$, and interpolate them on K to obtain $\hat{x}_A$ and $\hat{x}_B$.
5. Compute the quotient polynomial evaluation:
$$v_q := \frac{(v_a + \hat{x}_A(r))^2 - (v_b + \hat{x}_B(r))}{v_K(r)} \quad .$$
6. Check the following pairing equation:
$$e(T, \delta_2 H) \cdot e(U, \tau H) \cdot e(v_a \cdot \alpha G + v_b \cdot \beta G + v_q \cdot G - r \cdot U, H) \stackrel{?}{=} 0 \quad .$$

Figure 2: The unrolled PARI SNARK.

arkworks, and uses components from the HyperPlonk implementation.[6] Our implementation contributions are as follows:

Interface for programming GR1CS. We extend the R1CS programming interface in the ark-relations crate to also support programming GR1CS constraints in just 1500 lines of code. Our extension is user-friendly, and can take advantage of arkworks' strong library of R1CS constraint "gadgets".[7]

Implementation of GARUDA. We implement GARUDA atop ark-poly's multilinear polynomials. Our implementation supports multiple polynomial predicates, but we plan to extend it to support more general custom gates in the future.

Implementation of PARI. We implement PARI atop ark-poly's univariate polynomials. We additionally provide an adapter that converts R1CS constraints to SR1CS constraints. We also provide a Solidity code generator for PARI's verifier that on input the SR1CS instance size, outputs a Solidity contract that can be readily deployed on Ethereum.

Implementation of Polymath. We implement the Polymath SNARK [39] in Rust using the arkworks ecosystem with the same optimizations used for PARI.

8 Evaluation

Experimental setup. All measurements were conducted using a MacBook Pro with 18 GB of RAM and a 12-core Apple M3 Pro chip. For consistency across implementations, all benchmarks are single-threaded.

8.1 GARUDA

We investigate GARUDA's prover cost, verifier cost, and proof size. Our experiments run GARUDA on both R1CS and GR1CS, i.e. with and without custom gates.

Baselines. We compare against numerous baselines that represent the state-of-the-art in prover efficiency: (1) Groth16 [32], which supports free addition gates but not custom gates. (2) HyperPlonk[8] [17], which supports custom gates but not free addition gates. (3) Spartan [48], which supports R1CS, and SuperSpartan [49], which supports CCS. When the computation is uniform, the prover cost in both scales only with multiplication gates. However, when for non-uniform computations, cost scales with the *total* number of gates.

All SNARKs except (Super)Spartan use BLS12-381; the latter two use Curve25519. We note that, to our knowledge, there are no easily available implementations of SuperSpartan. Hence, in our experiments we approximate its prover cost by running the Spartan proof system on an R1CS instance that has the same total number of non-zero entries (= addition gates) and the same number of constraints as the GR1CS instance used for GARUDA. This simulation is realistic, and in fact favours SuperSpartan: instead of paying for a more complicated zerocheck which enforces custom gates, this simulation pays the cost of checking simpler R1CS constraints.

We also benchmark HyperPlonk using only high-degree custom gates, which is again favorable to HyperPlonk, as it excludes the overhead of addition or multiplication gates.

Experiments. We evaluate GARUDA's performance on three key metrics: prover time, verifier time, and proof size. We do so via both synthetic benchmarks (Sections 8.1.1 and 8.1.2), and via benchmarks on deployed applications (Section 8.1.3).

8.1.1 Prover time

To evaluate the performance of GARUDA's prover, we seek to answer three questions: (1) how does the prover's cost scale with increasing instance size? (2) what are the benefits of free addition gates? (3) what are the benefits of custom gates?

Scaling on synthetic circuits. In this experiment, we evaluated performance on a synthetic benchmark consisting of randomly-sampled R1CS instances of increasing size. The constraint systems were sampled as follows. Fix a maximum number of variables m in each constraint. Then, sample two m-sized linear-combinations (LCs) of variables uniformly at random, and enforce a multiplication constraint on these LCs.

The relevant baselines here are Spartan and Groth16, as these support R1CS. We excluded HyperPlonk, because it does not support R1CS.

The results are reported in the left plot of Fig. 3. The key takeaway is that GARUDA performs the best. It performs better than Spartan since addition gates are free, and performs better than Groth16 because it avoids G2 MSMs and FFTs.

Benefits of free addition gates. In this experiment, we evaluate the impact of free addition gates for non-uniform computations. We use the same setup and baselines as the previous experiment with the following tweaks. We fix the number of constraints to 2^{16}, and instead vary the number of variables in each constraint from 2 to 32. (This corresponds to varying the number of addition gates.)

The middle plot of Fig. 3 reports the results. As expected, prover time for both Groth16 and GARUDA does not vary as the number of addition gates increases, while Spartan's cost increases linearly. The gap grows quite large (almost 100×) when there are 32 variables in each constraint.

Benefits of custom gates. In this experiment, we evaluate the impact of custom predicates on prover time. Our benchmark circuit contains numerous iterations of the arithmetization-oriented Rescue-Prime hash function [50]. Over the scalar field of the BLS12-381 curve, the S-box exponent of Rescue-Prime α is 5 in the forward S-boxes.

Our GR1CS arithmetization of this circuit utilizes two types of local predicates: a standard R1CS predicate and a polynomial predicate of degree 5 that evaluates the exponentiation

[6] github.com/EspressoSystems/hyperplonk.

[7] github.com/arkworks-rs/r1cs-std and github.com/arkworks-rs/crypto-primitives.

[8] github.com/EspressoSystems/hyperplonk.

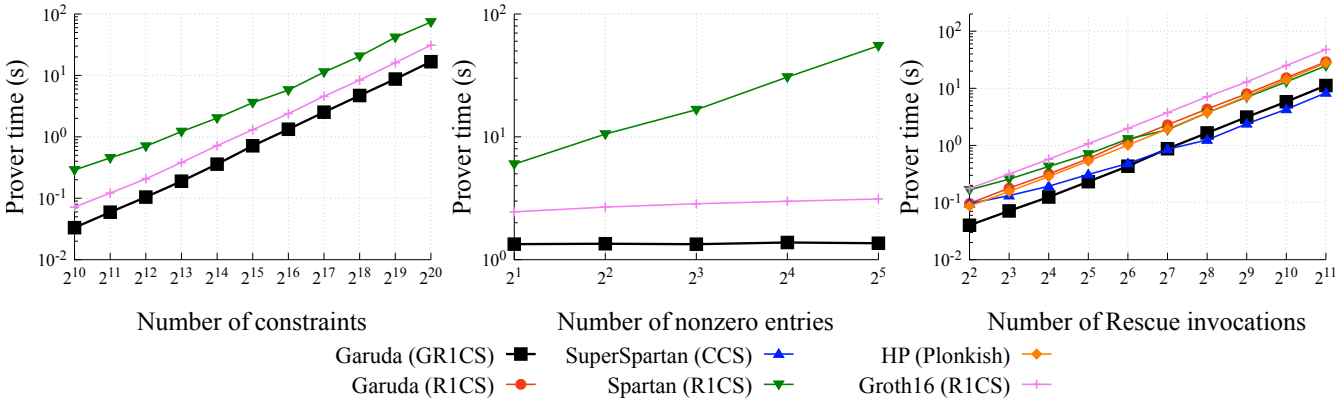

Figure 3: Comparison of prover time for GARUDA against the baselines. The left figure plots prover time for a synthetic random R1CS circuit. The middle plot demonstrates how cost scales as the number of addition gates increases. Finally, the right figure plots the effect of custom gates on prover time for the iterated-Rescue benchmark.

by α; the latter is used to check the correct evaluation of both the forward and inverse S-boxes.

We compare against all baselines here because HyperPlonk implements a benchmark circuit for Rescue-Prime. Since this circuit is uniform, the Spartan and SuperSpartan provers do not incur cryptographic costs for addition gates.

The right plot of Fig. 3 reports the results. In short, Garuda with custom gates (denoted Garuda-GR1CS) is consistently 4× faster than Groth16, and is 2.5× faster than Garuda over plain R1CS. However, SuperSpartan's cost is equal to or lower than Garuda's as it also supports both custom gates, and, for this uniform computation, free addition gates.

8.1.2 Verifier time and proof size

In all experiments from Section 8.1.1, GARUDA's verifier required roughly 3 ms to verify proofs. This is only slightly slower than Groth16 (2 ms), and is significantly faster than HyperPlonk (≈ 10 ms) and Spartan/SuperSpartan (10-1000ms).

Similarly, GARUDA's is concretely small, ranging from 5 kB to 8 kB. This is quite a bit larger than Groth16 (192 bytes), but is smaller than HyperPlonk (13 kB to 19 kB) and Spartan/SuperSpartan (20 kB to 50 kB).

8.1.3 Application benchmarks

To further evaluate the practical efficiency of GARUDA, we benchmark it on two real-world blockchain applications: Aptos Keyless [2] and Anon Aadhaar [1]. These applications represent settings where zkSNARK provers are executed on lightweight client devices. We compare the zero-knowledge instantiation of GARUDA against Spartan and Groth16, noting that current deployments of both applications rely on Groth16.

As shown in the following table, GARUDA achieves lower proving times than Groth16 on both circuits, reducing prover latency by 57.8% on Aptos Keyless (from 11.6s to 4.9s) and by 64.4% on Anon Aadhaar (from 10.4s to 3.7s). Spartan is significantly slower as the computations are non-uniform.

zkSNARK	Aptos Keyless	Anon Aadhaar
Spartan	162s	156s
Groth16	11.6s	10.4s
Garuda	4.9s	3.7s

8.2 PARI

To assess the efficiency benefits of PARI, we compared it against two state-of-the-art baselines: Polymath [39] and Groth16[9] [32]. For benchmarking verifier gas costs, we also compared PARI against the FFLONK smart contract[10] [27].

Number of constraints. Both PARI and Polymath use Square R1CS (SR1CS), whereas Groth16 uses R1CS. Converting R1CS to SR1CS increases the number of constraints by a constant factor of at most 2.

Proof size. PARI's proof comprises 2 $\mathbb{G}_1$ elements and 2 field elements. In 128-bit-secure curves, this is always smaller than that of Groth16 (2 $\mathbb{G}_1$ elements and 1 $\mathbb{G}_2$ element) and Polymath (3 $\mathbb{G}_1$ elements and 1 field element).

Scalability of proving. Fig. 4 shows the proving latency of PARI and non-zk variant of Groth16 over BLS12-381. As expected, for the same number of R1CS constraints, PARI incurs a higher ($\sim 2\times$) proving latency than Groth16, and Polymath incurs an even higher cost of $\sim 4\times$ that of Groth16.

Verifier time. As shown in Fig. 4, for an input size of 1, Groth16 achieves the fastest verification time ($\sim$ 0.8 ms), while PARI incurs slightly higher but still small costs ($\sim$ 0.9 ms). Polymath, in contrast, has a higher verification time of around 1.1 ms. However, as the input size increases, the verification time of PARI and Polymath remains nearly constant, whereas Groth16's verification time grows significantly. This is because, for large input sizes, Groth16's verification time

[9] github.com/arkworks-rs/groth16.

[10] https://github.com/kiwi202202/circom-gas-test.

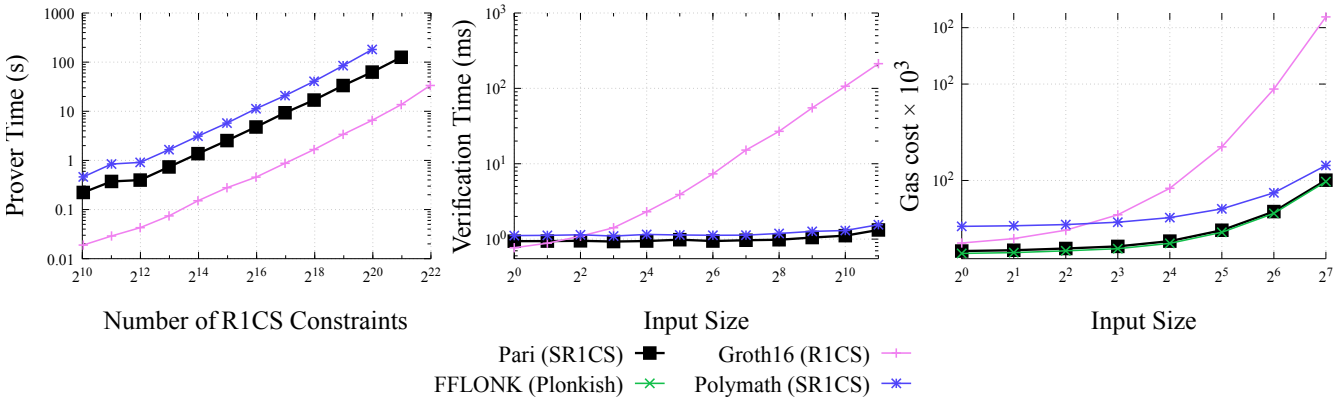

Figure 4: Comparison of the prover time, verification time, and gas cost for PARI, Groth16, and Polymath over BLS12-381

is dominated by a MSM of size equal to the input, whereas PARI and Polymath need much cheaper field operations.

Gas cost. For an input size of 1, PARI incurs a gas cost of 180,659, which is $\sim 6\%$ lower than Groth16 (191,396) and $\sim 17\%$ lower than FFLONK (215,720), but is about $\sim 1.6\%$ higher than Polymath (177,597). As the input size increases, the gas costs of PARI, FFLONK, and Polymath all grow at the same rate (~ 900 per additional public input), whereas Groth16's gas cost increases at a significantly higher rate ($\sim 7,160$ per additional public input).

9 Related work

We begin by first comparing our use of linear-subspace SNARKs for equifficiency to their use in LegoSNARK [16]. In brief, when limited to proving equifficiency for two committed polynomials, the two approaches are identical. However, they differ when the number of commitments increases. This is because LegoSNARK uses LS SNARKs to prove that one committed message is the concatenation of other committed messages while we use it to show that all committed messages are equal. This affects the handling of hiding as well.

9.1 SNARKs with small proof size

Groth16 [32] achieves the smallest known SNARK proof size with just three group elements. PARI improves upon this with proofs consisting of two group elements and two scalar field elements, and is always no larger than Groth16 (indeed strictly smaller when instantiated over BLS12-381) though it loses rerandomizability due to its use of Fiat–Shamir transformation. Moreover, while Groth16's verifier requires an MSM linear in the public input size, PARI performs only field operations of comparable size, making it more scalable for large inputs. Hashing the public input could help reduce the Groth16 verifier's group operations, but it is not free: a non-SNARK-friendly hash (e.g., SHA-256) increases prover costs, while a SNARK-friendly hash (e.g., Poseidon) increases verifier costs. GARUDA/PARI avoid this trade-off entirely.

Recent work. Polymath [39] is a SNARK consisting of three $\mathbb{G}_1$ elements and one $\mathbb{F}$ element. Over BLS12-381, Polymath proofs require 1408 bits, smaller than Groth16 proofs (1536 bits), but larger than PARI proofs (1280 bits). Polymath implicitly uses similar checks as PARI, but does not formalize these into a separate primitive as we do, and also does not support commitment batching. On the other hand, Polymath supports zero-knowledge, and proves security directly in the AGM with oblivious sampling (AGMOS) [40], whereas we prove security in the AGM. While we believe that PARI can be extended to support zero-knowledge and can be proven secure in the AGMOS, we leave this for future work.

Other related work. Barta et al. [3] construct a designated-verifier SNARK that achieves two-group elements in the designated verifier setting, but by relaxing their soundness error to be non-negligible. They also construct a single group element SNARK by relying on a non-standard assumption, and by settling for non-negligible completeness error.

Groth [32] and Groth and Maller [34] note that applying this idea to groups with *Type-I* pairings results in a two-group element SNARK. This work focuses on the more challenging setting of Type-III pairings, for which no prior two-group element SNARK was known.

Finally, a line of work [11, 12] constructs succinct designated-verifier proofs from lattice assumptions. Nitulescu [43] constructs an LIP for SAPs [34] that requires two queries, and showed that using this LIP in the methodology of Bitansky et al. [9], the resulting SNARK achieves a proof size of two lattice ciphertexts.

9.2 SNARKs supporting custom gates

TurboPlonk [26] introduced the concept of "high-degree" polynomial gates in circuits, while Plookup [28] introduced "lookup" gates. Both works require quasilinear prover time. Setty, Thaler, and Wahby [49] introduced a generalization of R1CS that they call "Customizable Constraint Systems"

(CCS), and described how to generalize Spartan [48] to support CCS. The resulting SNARK, called SuperSpartan, also achieves linear prover time. Our generalization of R1CS (GR1CS) subsumes both CCS and its variant CCS+ which additionally supports lookup gates, and we find it easier to work with. HyperPlonk [17] supports custom gates while providing linear prover time. As noted in Section 1, for non-uniform computations, all the foregoing works must pay cryptographic proving costs (e.g., group scalar multiplications) for linear gates. In contrast, GARUDA avoids these costs.

Ethical Considerations

This work presents new cryptographic constructions, GARUDA and PARI that improve prover efficiency and reduce proof size for SNARKs. As a theoretical and systems-oriented contribution to cryptography, our research does not involve human subjects, personal or sensitive data, or experiments on live or deployed systems. All evaluations are conducted on synthetic benchmarks and publicly described circuits, and the work introduces no new attack surfaces beyond those already present in standard cryptographic assumptions.

Stakeholders and impact. The primary stakeholders of this work are cryptography researchers, protocol designers, and developers of systems that rely on verifiable computation. The expected benefits are improved efficiency, reduced costs, and expanded design flexibility for SNARK-based systems, which may enable broader adoption of verifiable and privacy-preserving computation in benign applications such as verifiable computation, auditing, and integrity verification. End users may indirectly benefit from systems that become more efficient or accessible as a result of these improvements.

Dual-use considerations. As with many advances in cryptography, our techniques are dual use. More efficient and smaller SNARKs may be incorporated into systems whose downstream applications raise societal concerns, such as certain blockchain-based financial systems or privacy-enhancing technologies that could be misused to obscure illicit activity. These risks are not unique to our constructions and are inherent to general-purpose cryptographic primitives. Our work does not target any specific application domain, nor does it meaningfully lower barriers to misuse beyond incremental efficiency gains typical of research in this area.

Research and publication decisions. We chose to publish this work openly and with full technical detail because peer-reviewed dissemination enables scrutiny, validation, and responsible reuse by the research community. Transparency allows limitations, assumptions, and trade-offs, such as the requirement of circuit-specific trusted setup to be clearly understood and debated. We believe that the benefits of advancing the foundations of verifiable computation outweigh the indirect risks associated with dual use, and that open publication best supports long-term responsible development in cryptography.

Overall, this work adheres to established ethical norms in cryptographic research and contributes technical advances while acknowledging their broader context and implications.

Open Science

All artifacts needed to evaluate our work are publicly available at https://zenodo.org/records/17970155. Our library is structured as a Rust workspace. It contains:

- Implementations of GARUDA and PARI.
- Benchmark harnesses for our experiments.
- Comparative baselines, including both third-party SNARK libraries imported as Rust crates and our own re-implementations when crates were unavailable.

The benchmarks cover prover time, verifier time, and proof size, and reproduce all results reported in the paper. The repository includes build instructions and scripts to run experiments on different parameter sizes.

References

[1] Anon Aadhaar Project. "Anon Aadhaar: Privacy-Preserving Aadhaar Authentication". https://github.com/anon-aadhaar/anon-aadhaar. Accessed: 2025-06-06.

[2] Aptos Labs. "Keyless ZK Proofs". https://github.com/aptos-labs/keyless-zk-proofs. Accessed: 2025-06-06.

[3] O. Barta, Y. Ishai, R. Ostrovsky, and D. J. Wu. "On Succinct Arguments and Witness Encryption from Groups". In: CRYPTO '20.

[4] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina Melé. "Circom: A Circuit Description Language for Building Zero-Knowledge Applications". In: *IEEE Trans. Dependable Secur. Comput.* (2023).

[5] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: S&P '14.

[6] E. Ben-Sasson, A. Chiesa, L. Goldberg, T. Gur, M. Riabzev, and N. Spooner. "Linear-Size Constant-Query IOPs for Delegating Computation". In: TCC '19.

[7] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. "Aurora: Transparent Succinct Arguments for R1CS". In: EUROCRYPT '19.

[8] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. "Scalable Zero Knowledge via Cycles of Elliptic Curves". In: CRYPTO '14.

[9] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. "Succinct Non-Interactive Arguments via Linear Interactive Proofs". In: TCC '13.

[10] D. Boneh and X. Boyen. "Short Signatures Without Random Oracles". In: EUROCRYPT '04.

[11] D. Boneh, Y. Ishai, A. Sahai, and D. J. Wu. "Lattice-Based SNARGs and Their Application to More Efficient Obfuscation". In: EUROCRYPT '17.

[12] D. Boneh, Y. Ishai, A. Sahai, and D. J. Wu. "Quasi-Optimal SNARGs via Linear Multi-Prover Interactive Proofs". In: EUROCRYPT '18.

[13] S. Bowe, A. Gabizon, and I. Miers. "Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model". IACR ePrint Report 2017/1050.

[14] B. Bünz, B. Fisch, and A. Szepieniec. "Transparent SNARKs from DARK Compilers". In: EUROCRYPT '20.

[15] M. Campanelli, A. Faonio, D. Fiore, A. Querol, and H. Rodriguez. "Lunar: a Toolbox for More Efficient Universal and Updatable zkSNARKs and Commit-and-Prove Extensions". In: ASIACRYPT '21.

[16] M. Campanelli, D. Fiore, and A. Querol. "LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs". In: CCS '19.

[17] B. Chen, B. Bünz, D. Boneh, and Z. Zhang. "HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates". In: EUROCRYPT '23.

[18] A. Chiesa, M. A. Forbes, and N. Spooner. "A Zero Knowledge Sumcheck and its Applications". IACR ePrint Report 2017/305.

[19] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS". In: EUROCRYPT '20.

[20] arkworks contributors. `arkworks` *zkSNARK ecosystem*. 2022.

[21] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. "Multiparty Computation from Somewhat Homomorphic Encryption". In: CRYPTO '12.

[22] T. Datta, B. Chen, and D. Boneh. "VerITAS: Verifying Image Transformations at Scale". In: ed. by M. Blanton, W. Enck, and C. Nita-Rotaru. IEEE. DOI: 10.1109/SP61157.2025.00097.

[23] A. Escala, G. Herold, E. Kiltz, C. Ràfols, and J. Villar. "An Algebraic Framework for Diffie-Hellman Assumptions". Cryptology ePrint Archive, Paper 2013/377.

[24] A. Fiat and A. Shamir. "How to prove yourself: practical solutions to identification and signature problems". In: CRYPTO '86.

[25] G. Fuchsbauer, E. Kiltz, and J. Loss. "The Algebraic Group Model and its Applications". In: CRYPTO '18.

[26] A. Gabizon and Z. J. Williamson. "The turbo-plonk program syntax for specifying snark programs". Preprint.

[27] A. Gabizon and Z. J. Williamson. "fflonk: a Fast-Fourier inspired verifier efficient version of PlonK". IACR ePrint Archive, Paper 1167/2021.

[28] A. Gabizon and Z. J. Williamson. "plookup: A simplified polynomial protocol for lookup tables". IACR ePrint Report 2020/315.

[29] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge". IACR ePrint Report 2019/953.

[30] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. "Quadratic Span Programs and Succinct NIZKs without PCPs". In: EUROCRYPT '13.

[31] C. Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: STOC '09.

[32] J. Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: EUROCRYPT '16.

[33] J. Groth. "Short Pairing-Based Non-interactive Zero-Knowledge Arguments". In: ASIACRYPT '10.

[34] J. Groth and M. Maller. "Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs". In: CRYPTO '17.

[35] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. "Efficient Arguments without Short PCPs". In: CCC '07.

[36] A. Kate, G. M. Zaverucha, and I. Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: ASIACRYPT '10.

[37] E. Kiltz and H. Wee. "Quasi-Adaptive NIZK for Linear Subspaces Revisited". In: EUROCRYPT '15.

[38] V. Kolesnikov and T. Schneider. "Improved Garbled Circuit: Free XOR Gates and Applications". In: ICALP '08.

[39] H. Lipmaa. "Polymath: Groth16 Is Not The Limit". In: CRYPTO '24.

[40] H. Lipmaa, R. Parisella, and J. Siim. "Algebraic Group Model with Oblivious Sampling". In: TCC '23.

[41] T. Liu, T. Xie, J. Zhang, D. Song, and Y. Zhang. "Pianist: Scalable zkRollups via Fully Distributed Zero-Knowledge Proofs". In: IEEE S&P '24.

[42] T. Liu, X. Xie, and Y. Zhang. "zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy". In: ed. by Y. Kim, J. Kim, G. Vigna, and E. Shi. ACM. DOI: 10.1145/3460120.3485379.

[43] A. Nitulescu. "Lattice-Based Zero-Knowledge SNARGs for Arithmetic Circuits". In: LATINCRYPT '19.

[44] C. Papamanthou, E. Shi, and R. Tamassia. "Signatures of Correct Computation". In: TCC '13.

[45] T. P. Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: CRYPTO '91.

[46] W. Qu, Y. Sun, X. Liu, T. Lu, Y. Guo, K. Chen, and J. Zhang. "zkGPT: An Efficient Non-interactive Zero-knowledge Proof Framework for LLM Inference". In: *IACR Cryptol. ePrint Arch.* (2025).

[47] J. T. Schwartz. "Fast Probabilistic Algorithms for Verification of Polynomial Identities". In: *JACM* (1980).

[48] S. Setty. "Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup". In: CRYPTO '20.

[49] S. Setty, J. Thaler, and R. Wahby. "Customizable constraint systems for succinct arguments". ePrint Report 2023/552.

[50] A. Szepieniec, T. Ashur, and S. Dhooghe. "Rescue-Prime: a Standard Specification (SoK)". Cryptology ePrint Archive, Paper 2020/1143.

[51] B. WhiteHat. "roll_up: A Scalable Zero Knowledge Roll Up". Accessed: 2024-02-10.

[52] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. "Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation". In: CRYPTO '19.

[53] C. Zhang, Z. DeStefano, A. Arun, J. Bonneau, P. Grubbs, and M. Walfish. "Zombie: Middleboxes that Don't Snoop". In: ed. by L. Vanbever and I. Zhang. USENIX Association.

[54] R. Zippel. "Probabilistic algorithms for sparse polynomials". In: EUROSAM '79.

A Handling multiple predicates

To support multiple predicates $L_1,\ldots,L_t$, we propose to "stack" the matrices for different predicates together to get a single set of matrices $M_1^*,\ldots,M_t^*$. Then, the prover's oracles consist of the multilinear polynomials $\hat{z}_1,\ldots,\hat{z}_t$ that interpolate the matrix-vector products $M_i^*\cdot z$. To enforce the j-th local predicate L_j, the PIOP leverages *selector polynomials* S_j that activate the entries of $M_i^*\cdot z$ that correspond to the j-th predicate. That is, instead of directly checking $L_j(\hat{z}_1,\ldots,\hat{z}_t)=0$ (which would not work), we instead use the polynomials $S_j\cdot\hat{z}_1,\ldots,S_j\cdot\hat{z}_t$.[11] The satisfaction of each local predicate L_j is then enforced by a separate PIOP for rowcheck that checks that the polynomials $S_j\cdot\hat{z}_1,\ldots,S_j\cdot\hat{z}_t$ satisfy the predicate at all points in $\mathcal{D}$.[12]

In the case where there is only a single local predicate (e.g., in R1CS), there is no need for stacking and using selectors. This reduces proof size and (slightly) improves prover time complexity.

[11] In fact, when L_j is a polynomial predicate, we can leverage a better strategy that avoids doubling the individual degree of each polynomial by simply applying the selector to the entire expression $L_j(\hat{z}_1,\ldots,\hat{z}_t)$.

B Preliminaries

We denote by $\lambda\in\mathbb{N}$ a security parameter, and $\mathrm{negl}(\lambda)$ denotes an unspecified function that is *negligible* in λ.

We denote by $[n]$ the set $\{1,\ldots,n\}\subseteq\mathbb{N}$. We use $\boldsymbol{a}=[a_i]_{i=1}^n$ as a shorthand for the tuple/list $(a_1,\ldots,a_n)$; $|\boldsymbol{a}|$ denotes the number of entries in $\boldsymbol{a}$. If M is a matrix then $\|M\|$ denotes the number of nonzero entries in M. If each element v_i in the list $\boldsymbol{v}$ is itself a vector, then $\boldsymbol{v}[j]$ denotes the list containing the j-th element of each vector, i.e., $\boldsymbol{v}[j]=[v_i[j]]_{i=1}^n$.

If f is a function, then $[\![f]\!]$ denotes an oracle for the function. Also, we sometimes abuse the notation and write $[\![\boldsymbol{f}]\!]$ to denote the list of oracles $[\![f_i]\!]_{f_i\in\boldsymbol{f}}$.

We generally denote variables by the uppercase letters $X_1,\ldots,X_n$, while values that the variables can take on are denoted by the lowercase letters $x_1,\ldots,x_n$.

Random oracles. We denote by $\mathcal{U}_\lambda$ the set of all functions that map $\{0,1\}^*$ to $\{0,1\}^\lambda$. We denote by $\mathcal{U}_*$ the set $\bigcup_{\lambda\in\mathbb{N}}\mathcal{U}_\lambda$. A *random oracle* with security parameter λ is a function $\rho\colon\{0,1\}^*\to\{0,1\}^\lambda$ sampled uniformly at random from $\mathcal{U}_\lambda$.

B.1 Algebraic background

For a D-dimensional polynomial vector space $\mathbb{K}$ over a field $\mathbb{F}$, we denote by $\mathscr{B}$ a basis of $\mathbb{K}$. A *punctured basis* $\mathscr{P}=(\mathcal{J},\mathscr{B})$ is any subset of $\mathscr{B}$, where $\mathcal{J}\subset[D]$ is an indexing set that enumerates the basis elements of $\mathscr{B}$ appearing in $\mathscr{P}$. We say that a polynomial p is expressible in a punctured basis $\mathscr{P}$ if it can be written as $p=\sum_{j\in\mathcal{J}}a_j\cdot b_j$ for coefficients $a_j\in\mathbb{F}$ and basis elements $b_j\in\mathscr{B}$. As before, we denote by $[p]_{\mathscr{P}}$ the coefficient vector of p when expressed in $\mathscr{P}$. We emphasize that p is expressible in a punctured basis $\mathscr{P}$ if and only if its coefficients with respect to basis elements b_j for $j\notin\mathcal{J}$ are zero.

B.1.1 Univariate polynomials

A smooth subgroup K of the multiplicative group $\mathbb{F}^*$ of a finite field $\mathbb{F}$ is a subgroup with size equal to a power of two. We associate with such subgroups an ordering ϕ_K that is a bijection from K to the set $\{0,1,\ldots,|K|-1\}$.

Polynomial encodings. For a finite field $\mathbb{F}$, multiplicative subgroup $K\subseteq\mathbb{F}^*$, and function $f\colon[|K|]\to\mathbb{F}$, we denote by $\hat{f}$ the (unique) univariate polynomial over $\mathbb{F}$ with degree less than $|K|$ such that $\hat{f}(a)=f(\phi_K(a))$ for every $a\in K$.

[12] In practice, PIOPs for separate predicates can again be batched together to reduce the prover work and proof size.

Vanishing polynomials. Every element of a subgroup K vanishes on the polynomial $v_K(X) = X^{|K|} - 1$, and v_K can be evaluated in $O(\log |K|)$ field operations.

Lagrange polynomials. For a subgroup K and element $a \in K$, $\mathcal{L}_a^K$ denotes the unique polynomial of degree less than $|K|$ such that $\mathcal{L}_a^K(a) = 1$ and $\mathcal{L}_a^K(b) = 0$ for all $b \in K \setminus \{a\}$.

B.1.2 Multivariate and multilinear polynomials

The boolean hypercube $\{0,1\}^\nu$ is denoted by $\mathcal{B}_\nu$.

Multilinear extension (MLE). A multivariate polynomial $p(\boldsymbol{X})$ is *multilinear* if each variable has individual degree 1. Any function $f : \mathcal{B}_\nu \to \mathbb{F}$ has a unique *multilinear extension* defined as the polynomial $\hat{f}(\boldsymbol{X})$ such that $\hat{f}$ agrees with f on all points in $\mathcal{B}_\nu$. For a given point $\alpha \in \mathcal{B}_\nu$, the Lagrange multilinear basis polynomial $\mathrm{eq}(\alpha, \boldsymbol{X})$ is defined as $\mathrm{eq}(\alpha, \boldsymbol{X}) := \prod_{i=1}^{\nu}(1 - (1 - 2\alpha_i)X_i)$. The multilinear extension of any function $f : \mathcal{B}_\nu \to \mathbb{F}$ can be computed in $O(2^\nu)$ operations as $\hat{f}(\boldsymbol{X}) := \sum_{\alpha \in \mathcal{B}_\nu} f(\alpha) \cdot \mathrm{eq}(\alpha, \boldsymbol{X})$.

We denote the 2^ν-dimensional vector space of multilinear polynomials over $\mathbb{F}$ by $\mathrm{Mult}(\mathbb{F}[X_1, \ldots, X_\nu])$.

B.2 Cryptographic assumptions

The cryptographic primitives that we construct rely on cryptographic assumptions about bilinear groups. We formalize these via a *bilinear group sampler*, which is a probabilistic polynomial-time algorithm $\mathsf{SampleGrp}$ that, on input a security parameter λ (represented in unary), outputs a tuple $\langle \mathsf{group} \rangle = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, G, H, e)$ where $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are cyclic groups of a prime order $q \in \mathbb{N}$, G generates $\mathbb{G}_1$, H generates $\mathbb{G}_2$, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a (non-degenerate) bilinear map.

B.2.1 Strong Diffie–Hellman

Assumption 1 ([10]). The Strong Diffie–Hellman (SDH) assumption states that for every efficient adversary $\mathcal{A}$ and degree bound $d \in \mathbb{N}$, the following probability is negligible in the security parameter λ:

$$\Pr\left[C = \frac{1}{\beta + c} G \;\middle|\; \begin{array}{r} \langle \mathsf{group} \rangle \leftarrow \mathsf{SampleGrp}(1^\lambda) \\ \beta \leftarrow \mathbb{F}_q \\ \boldsymbol{\Sigma} \leftarrow \{\{\beta^i G\}_{i=0}^d, \beta H\} \\ (c, C) \leftarrow \mathcal{A}(\langle \mathsf{group} \rangle, \boldsymbol{\Sigma}) \end{array} \right] .$$

B.2.2 Algebraic group model

In order to achieve additional efficiency, we construct PC schemes in the Algebraic Group Model (AGM) [25] which replaces specific knowledge assumptions (such as Power Knowledge of Exponent [33] assumptions).

Definition B.1 (algebraic algorithm)**.** *Let $\mathbb{G}$ be cyclic group of prime order q and $\mathcal{A}_{\mathsf{alg}}$ a probabilistic algorithm. During its execution $\mathcal{A}_{\mathsf{alg}}$ may interact with oracles or other parties and receive further inputs including obliviously sampled group elements (which it cannot sample directly). Let $\boldsymbol{L} \in \mathbb{G}^n$ be the list of all group elements $\mathcal{A}_{\mathsf{alg}}$ has been given so far such that all other inputs it has received do not depend in any way on group elements. We call $\mathcal{A}_{\mathsf{alg}}$ algebraic if whenever it outputs a group element $G \in \mathbb{G}$ it also outputs a vector $\boldsymbol{a} = [a_i]_{i=1}^n \in \mathbb{F}_q^n$ such that $G = \sum_{i=1}^n a_i L_i$. The coefficients $\boldsymbol{a}$ are called the "representation" of G with respect to $\boldsymbol{L}$.*

B.3 Indexed relations

In this work, we will consider constraint systems that support a rich variety of constraints. We begin by formalizing these constraints first, and then describe our new constraint system relation, which we call the *Generalized Rank-1 Constraint System* (GR1CS) relation.

Definition B.2 (local predicate)**.** *Given a finite field $\mathbb{F}$ and an input arity $t \in \mathbb{N}$, a* **local predicate** *L is a function $L : \mathbb{F}^t \to \{0,1\}$. An input $x \in \mathbb{F}^t$ is said to* **satisfy** *L if $L(x) = 0$.*

Examples of local predicates include:

- *Polynomial predicate P* defined by a multivariate polynomial $p \in \mathbb{F}[X_1, \ldots, X_t]$, so that $P(X_1, \ldots, X_t) = 0$ if and only if $p(X_1, \ldots, X_t) = 0$.
- *Table membership predicate T* defined by a finite subset $\mathcal{T} \subset \mathbb{F}^t$, so that $T(X_1, \ldots, X_t) = 0$ if and only if $(X_1, \ldots, X_t) \in \mathcal{T}$.

For simplicity, we will stipulate that all the local predicates considered in this paper have the same number of input variables n, same number of total variables k, and same arity t. This is without loss of generality, as one can always pad the arity of the predicates with dummy variables.

Definition B.3 (generalized R1CS)**.** *The indexed relation $\mathcal{R}_{\mathrm{GR1CS}}$ is the set of all triples $(\mathbb{i}, \mathbb{x}, \mathbb{w}) = \big((\mathbb{F}, n, k, m, c, t, \mathscr{C}), x, w\big)$ where:*

- *$\mathbb{F}$ is a finite field,*
- *n is the number of public input (instance) variables,*
- *k is the total number of variables in the constraint system,*
- *c is the number of constraint sets.*
- *t is the arity of the predicates*
- *$\mathscr{C} = [C_i = (L_i, \boldsymbol{M}_i, m_i)]_{i=1}^c$ are custom constraints s.t.:*
  - *$\boldsymbol{M}_i := M_{i,1}, \ldots, M_{i,t} \in \mathbb{F}^{m_i \times k}$ are* constraint matrices*,*
  - *L_i is a local predicate (Definition B.2).*

A triple is in $\mathcal{R}_{\mathrm{GR1CS}}$ if, for each $(L_i, \boldsymbol{M}_i, m_i) \in \mathscr{C}$ and each $j \in [m_i]$, and with $z := (x, w) \in \mathbb{F}^k$, it holds that

$$L((M_{i,1} z)[j], \ldots, (M_{i,t} z)[j]) = 0 \quad .$$

R1CS is a special case of GR1CS where $t = 3$ and L is the quadratic polynomial predicate $L(a, b, c) = ab - c$. We will also consider *Square R1CS* [34], which is a special case of GR1CS where $t = 2$ and $L(a, b) = a^2 - b$. Groth and Maller [34] show that R1CS can be reduced to SR1CS, thus making SR1CS NP-complete.

Remark B.4. *In this paper, we will consider instances of GR1CS where the matrices obtained by concatenating the*

constraint matrices all have rank at least equal to the number of variables in the GR1CS instance. This is without loss of generality, as one can always add a small number of dummy constraints to ensure this property. This requirement arises due to technicalities in the knowledge-soundness proof of our SNARK construction.

B.4 Polynomial commitment schemes

A polynomial commitment scheme PC must satisfy the following completeness and extractability properties, and optionally also the following hiding.

Completeness. For every maximum bound $D \in \mathbb{N}$ and efficient adversary $\mathcal{A}$:

$$\Pr\left[\begin{array}{c|r} \dim(p) \le D & (\mathsf{ck},\mathsf{ok},\mathsf{vk}) \leftarrow \mathsf{PC.Setup}(1^\lambda, D) \\ \Downarrow & (p,u) \leftarrow \mathcal{A}(\mathsf{ck},\mathsf{ok}) \\ \mathsf{PC.Verify}(\mathsf{vk}, & c \leftarrow \mathsf{PC.Commit}(\mathsf{ck},p) \\ c,u,v,\pi) = 1 & \pi \leftarrow \mathsf{PC.Open}(\mathsf{ok},p,u) \\ & v := p(u) \end{array}\right] = 1 \ .$$

Extractability. For every maximum bound $D \in \mathbb{N}$ and efficient adversary $\mathcal{A}$, there exists an efficient extractor $\mathcal{E}$ such that for every round bound $\mathsf{r} \in \mathbb{N}$, every efficient public-coin challenger $\mathcal{C}$, efficient query sampler Q, and efficient adversary $\mathcal{B}$, the following probability is overwhelming:

$$\Pr\left[\begin{array}{c|r} \forall i \in \{1,\ldots,\mathsf{r}\}: & srs \leftarrow \mathsf{EPC.Setup}(1^\lambda, D) \\ \mathsf{PC.Verify}(\mathsf{vk}, c_i, u, & \text{where } srs = (\mathsf{ck},\mathsf{ok},\mathsf{vk}) \\ v_i, \pi_i) = 1 & \hline \\ \Downarrow & \text{For } i \in \{1,\ldots,\mathsf{r}\}: \\ \dim(p) \le D, & \rho_i \leftarrow \mathcal{C}(\mathsf{ck},\mathsf{ok},i) \\ p_i(u) = v_i & c_i \leftarrow \mathcal{A}(\mathsf{ck},\mathsf{ok},[\rho_j]_{j=1}^{i}) \\ & p_i \leftarrow \mathcal{E}(\mathsf{ck},\mathsf{ok},[\rho_j]_{j=1}^{i}) \\ & \hline \\ & u \leftarrow Q(\mathsf{ck},\mathsf{ok},[c]_{i=1}^{\mathsf{r}},[\rho_i]_{i=1}^{\mathsf{r}}) \\ & \mathcal{B}(u) \leftarrow ([v_i]_{i=1}^{\mathsf{r}},[\pi_i]_{i=1}^{\mathsf{r}}) \end{array}\right] .$$

Hiding. There exists a polynomial-time simulator $\mathcal{S} = (\mathsf{Setup},\mathsf{Commit},\mathsf{Open})$ such that, for every dimension bound $D \in \mathbb{N}$, round bound $\mathsf{r} \in \mathbb{N}$, and unbounded adversary $\mathcal{A} = (\mathcal{A}_1,\mathcal{A}_2,\mathcal{A}_3,\mathcal{A}_4)$, the probability that $b = 1$ in the following two experiments is identical.

$\mathsf{Real}(1^\lambda, D, \mathcal{A})$:
1. $(\mathsf{ck},\mathsf{ok},\mathsf{vk}) \leftarrow \mathsf{EPC.Setup}(1^\lambda, D)$.
2. Letting $c_0 := \perp$, for $i = 1,\ldots,\mathsf{r}$:
   (a) $p_i \leftarrow \mathcal{A}_2(\mathsf{ck},\mathsf{ok},c_0,c_1,\ldots,c_{i-1})$.
   (b) Sample randomness ω_i and compute the real commitment: $c_i \leftarrow \mathsf{PC.Commit}(\mathsf{ck},p_i;\omega_i)$.
3. $\boldsymbol{c} := [c_i]_{i=1}^{\mathsf{r}}$, $\boldsymbol{\omega} := [\omega_i]_{i=1}^{\mathsf{r}}$.
4. $u \leftarrow \mathcal{A}_3(\mathsf{ck},\mathsf{ok},\boldsymbol{c},\boldsymbol{\omega})$.
5. For $i = 1,\ldots,\mathsf{r}$, set $\pi_i \leftarrow \mathsf{PC.Open}(\mathsf{ck},p_i,u;\omega_i)$.
6. $\boldsymbol{\pi} := [\pi_i]_{i=1}^{\mathsf{r}}$, $\boldsymbol{v} := [p_i(u)]_{i=1}^{\mathsf{r}}$.
7. $b \leftarrow \mathcal{A}_4(\mathsf{vk},\boldsymbol{c},u,\boldsymbol{v},\boldsymbol{\pi})$.

$\mathsf{Ideal}(1^\lambda, D, \mathcal{A})$:
1. $(\mathsf{ck},\mathsf{ok},\mathsf{vk},\mathsf{trap}) \leftarrow \mathcal{S}.\mathsf{Setup}(1^\lambda, D)$.
2. Letting $c_0 := \perp$, for $i = 1,\ldots,\mathsf{r}$:
   (a) $p_i \leftarrow \mathcal{A}_2(\mathsf{ck},\mathsf{ok},c_0,c_1,\ldots,c_{i-1})$.
   (b) Sample randomness ω_i and compute the simulated commitment: $c_i \leftarrow \mathcal{S}.\mathsf{Commit}(\mathsf{trap};\omega_i)$.
3. $\boldsymbol{c} := [c_i]_{i=1}^{\mathsf{r}}$, $\boldsymbol{\omega} := [\omega_i]_{i=1}^{\mathsf{r}}$.
4. $u \leftarrow \mathcal{A}_3(\mathsf{ck},\mathsf{ok},\boldsymbol{c},\boldsymbol{\omega})$.
5. For $i = 1,\ldots,\mathsf{r}$:
   (a) $\pi_i \leftarrow \mathcal{S}.\mathsf{Open}(\mathsf{trap},p_i(u),u;\omega_i)$.
6. $\boldsymbol{\pi} := [\pi_i]_{i=1}^{\mathsf{r}}$, $\boldsymbol{v} := [p_i(u)]_{i=1}^{\mathsf{r}}$.
7. $b \leftarrow \mathcal{A}_4(\mathsf{vk},\boldsymbol{c},u,\boldsymbol{v},\boldsymbol{\pi})$.

B.5 Succinct arguments of knowledge

An *argument of knowledge* ARG for an indexed NP relation $\mathcal{R}$ is a tuple of algorithms $(\mathcal{G},\mathcal{P},\mathcal{V})$ that satisfies the following syntax and properties.

- The **generator** $\mathcal{G}$ is a probabilistic algorithm that takes as input a security parameter λ (in unary) and an index $\mathbb{i}$ and outputs the proving and verification keys $(\mathsf{ipk},\mathsf{ivk})$.
- The **prover** $\mathcal{P}$ is a probabilistic algorithm that takes as input the proving key ipk, an instance $\mathbb{x}$, and a witness $\mathbb{w}$ and outputs a proof π.
- The **verifier** $\mathcal{V}$ is a deterministic algorithm that takes as input the verification key ivk, an instance $\mathbb{x}$, and a proof π and outputs a bit indicating whether π is valid.

These above algorithms must satisfy standard completeness, knowledge-soundness, and optionally (perfect) zero knowledge properties.

B.6 Linear subspace SNARKs

Fix a bilinear group $(\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,G,H,e)$. Then a linear subspace SNARK for this group proves the following relation.

Definition B.5. *The indexed relation $\mathcal{R}_{\mathsf{LS}}$ is the set of all index-instance-witness triples $(\mathbb{i},\mathbb{x},\mathbb{w}) = (\boldsymbol{M},\boldsymbol{y},\boldsymbol{x})$ where $\mathbb{F}$ is a finite field, $\boldsymbol{M} \in \mathbb{G}_1^{n\times t}$ is a matrix of group elements, $\boldsymbol{y} \in \mathbb{G}_1^n$ is a vector of group elements in $\mathbb{G}_1$, and $\boldsymbol{x} \in \mathbb{F}^t$ is a vector of scalars, such that the equality $\boldsymbol{y} = \boldsymbol{M}\cdot\boldsymbol{x}$ holds.*

Kiltz and Wee [37] provide a zkSNARK for $\mathcal{R}_{\mathsf{LS}}$ under the k-Matrix Diffie-Hellmann Assumption [23], and Campanelli et. al. [16] prove it knowledge-sound in the AGM.

C Equifficient polynomial commitments

C.0.1 Extractability

Theorem C.1. *The construction* EPC *in Section 3.3.3 achieves equifficient extractability.*

Proof. Let $(\mathcal{A}_1,\mathcal{A}_2)$ be an efficient stateful adversary against EPC. We construct an efficient extractor $\mathcal{E}$ against $(\mathcal{A}_1,\mathcal{A}_2)$ that succeeds with overwhelming probability, such that for

every round bound $\mathsf{r} \in \mathbb{N}$, efficient public-coin sampler $\mathcal{C}$, query sampler Q, constraint sampler Ψ, and adversary $\mathcal{A}_3$, the following probability is negligibly close to 1:

Denote by $\boldsymbol{\Theta}$ the set of non-trivial constraints in $\boldsymbol{\Omega}$, and by W the first non-trivial constraint appearing in $\boldsymbol{\Omega}$. Also, denote by $\boldsymbol{\Lambda}_i \cap \boldsymbol{\Theta}$ the set of non-trivial constraints in $\boldsymbol{\Lambda}_i$, and by m_i the number of constraints output by the constraint sampler Ψ for round i.

For each round $i \in [\mathsf{r}]$, we construct the extractor $\mathcal{E}$ for EPC below and argue that it succeeds.

$\mathcal{E}(\mathsf{ck}, \mathsf{ok}, [\boldsymbol{\Lambda}_j]_{j=1}^{i}, [\mathsf{p}_j]_{j=1}^{i}) \to \boldsymbol{P}$:

1. Parse ck as $(\mathbb{K}, \langle \mathsf{group} \rangle, \boldsymbol{\Omega}, \mathsf{ck}_{\mathsf{PC}}, [\mathsf{ipk}_S]_{S \in \boldsymbol{\Lambda} \cap \boldsymbol{\Theta}})$, and get $\mathsf{ok}_{\mathsf{PC}}$ from the opener key ok.
2. Parse $\boldsymbol{\Lambda}_i$ as $[\Lambda_1, \ldots, \Lambda_{m_i}]_{j=1}^{m_i}$.
3. $[a_{\Lambda_j,1}, \ldots, a_{\Lambda_j,|\Lambda_j|}, \tilde{a}_{\Lambda_j}]_{j=1}^{m_i} \leftarrow \mathcal{A}_2(\mathsf{ck}, \mathsf{ok}, [\boldsymbol{\Lambda}_j]_{j=1}^{i}, [\mathsf{p}_j]_{j=1}^{i})$.
4. Forward $[a_{\Lambda_j,1}, \ldots, a_{\Lambda_j,|\Lambda_j|}]_{j=1}^{m_i}$ to $\mathcal{E}_{\mathsf{PC}}(\mathsf{ck}_{\mathsf{PC}}, \mathsf{ok}_{\mathsf{PC}}, [\mathsf{p}_j]_{j=1}^{i})$ and receive $[\mathsf{w}_{\Lambda_j,1}, \ldots, \mathsf{w}_{\Lambda_j,|\Lambda_j|}]_{j=1}^{m_i}$, or $\perp$ if $\mathcal{E}_{\mathsf{PC}}$ aborts. In the latter case, output $\perp$.
5. For each non-trivial constraint $\Lambda_j \in \boldsymbol{\Lambda}_i \cap \boldsymbol{\Theta}$, express the vector $\mathsf{w}_{\Lambda_j,\kappa}$ in the punctured basis $\mathscr{P}_\kappa \in \Lambda_j$, transforming it from its representation in the canonical basis $\mathscr{U}$. If this is not possible, output $\perp$. Otherwise, set $\boldsymbol{p}_j := [\langle \mathsf{w}'_{\Lambda_j,1}, \mathscr{P}_1 \rangle, \ldots, \langle \mathsf{w}'_{\Lambda_j,|\Lambda_j|}, \mathscr{P}_{|\Lambda_j|} \rangle]$, where $\mathsf{w}'_{\Lambda_j,\kappa}$ are the resulting vectors in the new basis.
6. For each trivial constraint $\Lambda_k \in \boldsymbol{\Lambda}_i \cap \boldsymbol{\Theta}$, set $\boldsymbol{p}_k := \langle \mathsf{w}_{\Lambda_k,1}, \mathscr{U} \rangle$.
7. Output the polynomials $\boldsymbol{P} := [\boldsymbol{p}_1, \ldots, \boldsymbol{p}_{m_i}]$.

The extractor $\mathcal{E}$ can fail with non-negligible probability due to at least one of three reasons. We analyze each case separately and argue it occurs with negligible probability, or that it contradicts our original assumptions.

1. *Incorrect evaluation*: there exists a polynomial $p_{i,j} \in \boldsymbol{P}$ whose evaluation is incorrect.
2. *Coefficient non-equality*: the equifficient property fails, i.e., there exists a round $i \in [\mathsf{r}]$ and a (non-trivial) constraint $\Lambda_j \in \boldsymbol{\Lambda}_i$ such that $\Lambda_j(\boldsymbol{p}_j) = 0$.
3. $\mathcal{E}$ *outputs* $\perp$: either $\mathcal{E}_{\mathsf{PC}}$ outputs $\perp$, or there is an extracted polynomial $p_{i,\kappa}$ that is not expressible in its associated punctured basis $\mathscr{P}_{i,\kappa} \in \Lambda_i$.

(1) Incorrect evaluation. This straightforwardly follows from the evaluation binding property of PC.

(2) Coefficient non-equality. We proceed by contradiction. Suppose that in round $i \in [\mathsf{r}]$, $\mathsf{EPC.Verify}$ accepts, the extractor $\mathcal{E}$ does not output $\perp$, and there exists a list of extracted polynomials $\boldsymbol{p}_j \in \boldsymbol{P}_i$ that do not satisfy their equifficient constraint; in particular, $\Lambda_j(\boldsymbol{p}_j) = 0$. Let $[c_1, \ldots, c_{|\Lambda_j|}]$ be the commitments from which the polynomials in $\boldsymbol{p}_j$ were extracted from, and denote this list by $\mathbb{x}_{\mathsf{LS}}$. Since $\mathsf{EPC.Verify}$ accepts, it follows that the verifier for the linear subspace argument Π_{LS} for Λ_j also accepts; that is, $\mathcal{V}_{\mathsf{LS}}(\mathsf{ivk}_{\mathsf{LS},\Lambda_j}, \mathbb{x}_{\mathsf{LS}}, \pi_{\mathsf{LS}}) = 0$. However, this occurs with negligible probability by the soundness property of Π_{LS}, as the polynomials committed to in $\mathbb{x}_{\mathsf{LS}}$ are equifficient if and only if they satisfy the linear subspace relation $\mathcal{R}_{\mathbb{L}\mathsf{S}}$.

(3) $\mathcal{E}$ outputs $\perp$. Suppose there exists a round $i \in [\mathsf{r}]$ where $\mathsf{EPC.Verify}$ accepts and the $\mathcal{E}$ outputs $\perp$, then either the extractor $\mathcal{E}_{\mathsf{PC}}$ for PC outputs $\perp$, or there is an extracted polynomial not expressible in its associated punctured basis. For the former, a union-bound over the probabilities that $\mathcal{E}_{\mathsf{PC}}$ aborts for any $a_{\Lambda_j,\kappa}$ in Step 4 is negligible by the knowledge-soundness property of PC. For the latter, a polynomial not expressible in its punctured basis would fail to satisfy the linear subspace relation as expressing it would require linear combinations of ambient basis elements not inside its punctured basis (in particular, these extra basis elements do not appear in $\mathbb{i}_{\mathsf{LS}}$). This occurs with negligible probability by the soundness of Π_{LS} whenever $\mathsf{EPC.Verify}$ accepts.

□

D EPC scheme for univariate polynomials

We describe a construction of EPC schemes for univariate polynomials that supports batched commitments.

D.0.1 Construction

Setup. $\mathsf{EPC.Setup}$ samples public parameters pp as follows. Sample a bilinear group $\langle \mathsf{group} \rangle \leftarrow \mathsf{SampleGrp}(1^\lambda, q)$ and parse $\langle \mathsf{group} \rangle$ as a tuple $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, G, H, e)$. Let $\mathbb{K}$ be the vector space $\mathbb{F}_q^{\leq D}[X]$ whose canonical basis $\mathscr{U}$ is the set of monomials $\{1, x, x^2, \ldots, x^{D-1}\}$. Sample $\tau \leftarrow \mathbb{F}_q$ uniformly at random, and compute the vector of monomial encodings:

$$\boldsymbol{\Sigma} := \left(G, \tau \cdot G, \tau^2 \cdot G, \ldots, \tau^{D-1} \cdot G\right) \in \mathbb{G}_1^D$$

Set $\mathsf{pp} := (\mathbb{K}, \langle \mathsf{group} \rangle, \boldsymbol{\Sigma}, \tau \cdot H)$ and output the public parameters pp. These public parameters will support univariate polynomials over the field $\mathbb{F}_q$ of degree at most $D-1$.

Specialize. Given oracle access to public parameters pp and input a set of equifficient constraints $\boldsymbol{\Omega} = \{\Omega_i\}_{i=1}^n$, $\mathsf{EPC.Specialize}$ will compute committer, opener, and verifier keys as follows.

Sample randomness as follows. For every non-trivial constraint $S \in \boldsymbol{\Omega}$, sample uniformly random field elements $\boldsymbol{\alpha}_S := (1, \alpha_S, \alpha_S^2, \ldots, \alpha_S^{|S|-1}) \leftarrow \mathbb{F}_q^{|S|}$ and $\gamma_S, \delta_S \leftarrow \mathbb{F}_q$. Then, for every trivial constraint $V \in \boldsymbol{\Omega}$, sample the uniformly random field element $\boldsymbol{\alpha}_V := (\alpha_V) \leftarrow \mathbb{F}_q$.

Construct the committer key ck as follows. First, for each non-trivial constraint $S \in \boldsymbol{\Omega}$,

1. Parse S as $[\mathscr{P}_1, \ldots, \mathscr{P}_{|S|}]$, and each $\mathscr{P}_i$ as $(\mathcal{J}_i, \mathscr{B}_i)$, and each $\mathscr{B}_i$ as the basis elements $\{b_s^{(i)}\}_{s=1}^D$ (expressed in the monomial basis).
2. For each $i \in [|S|]$, compute encodings of the basis elements in $\mathscr{P}_i$: $\boldsymbol{b}_S^{(i)} := (b_s^{(i)}(\tau) \cdot G)_{s \in \mathcal{J}_i} \in \mathbb{G}_1^{|\mathscr{P}_1|}$.
3. Construct the 'consistency' committer key ck_S for S by taking a linear combination of the vectors $\boldsymbol{b}_S^{(1)}, \ldots, \boldsymbol{b}_S^{(|S|)}$

with respect to the coefficients $\boldsymbol{\alpha}_S$:

$$\mathsf{ck}_S := \frac{\sum_{i=1}^{|S|} \alpha_S^{i-1} \cdot \boldsymbol{b}_S^{(i)}}{\delta_S} \in \mathbb{G}_1^{|\mathscr{P}_1|}$$

Next, to construct keys for trivial constraints, denote by W the first non-trivial constraint in $\boldsymbol{\Omega}$; commitments to polynomials with trivial constraints will be batched with commitments to polynomials constrained by W. (Note that W is not necessarily the first constraint Ω_1.) For every trivial constraint $V \in \boldsymbol{\Omega}$, compute the committer key for V as $\mathsf{ck}_V := \frac{\alpha_V}{\delta_W} \cdot \boldsymbol{\Sigma} \in \mathbb{G}_1^D$. (Recall that we assume that all trivial constraints are just the canonical monomial basis.) Finally, construct the committer key $\mathsf{ck} := (\mathbb{K}, \langle \mathsf{group} \rangle, \boldsymbol{\Omega}, \boldsymbol{\Sigma}, (\mathsf{ck}_T)_{T \in \boldsymbol{\Omega}})$.

Next, construct the opener key ok as follows. For each non-trivial constraint $S \in \boldsymbol{\Omega}$, compute the opener key for S as $\mathsf{ok}_S := (\frac{1}{\gamma_S} \cdot \boldsymbol{\Sigma}, \frac{\alpha_S}{\gamma_S} \cdot \boldsymbol{\Sigma}, \ldots, \frac{\alpha_S^{|S|-1}}{\gamma_S} \cdot \boldsymbol{\Sigma}) \in \mathbb{G}_1^{|S| \cdot D}$, while for every trivial constraint $V \in \boldsymbol{\Omega}$, construct the opener key as $\mathsf{ok}_V := \frac{\alpha_V}{\gamma_W} \cdot \boldsymbol{\Sigma} \in \mathbb{G}_1^D$. Set the opener key to be $\mathsf{ok} := (\mathbb{K}, \langle \mathsf{group} \rangle, \boldsymbol{\Omega}, \boldsymbol{\Sigma}, (\mathsf{ok}_T)_{T \in \boldsymbol{\Omega}})$.

Finally, construct the verifier key as follows. Denote by $\boldsymbol{\Theta}$ the set of non-trivial constraints in $\boldsymbol{\Omega}$. Then the verifier key vk is defined as $\mathsf{vk} := (\mathbb{K}, \langle \mathsf{group} \rangle, \boldsymbol{\Omega}, (\frac{\alpha_V \cdot G}{\gamma_W})_{V \in \boldsymbol{\Omega} \setminus \boldsymbol{\Theta}}, \{(\frac{\boldsymbol{\alpha}_S \cdot G}{\gamma_S}), (\gamma_S \cdot H), (\gamma_S \tau \cdot H), (\delta_S \cdot G)\}_{S \in \boldsymbol{\Theta}})$.

Commit. On input $\mathsf{ck} = (\mathbb{K}, \langle \mathsf{group} \rangle, \boldsymbol{\Omega}, \boldsymbol{\Sigma}, (\mathsf{ck}_T)_{T \in \boldsymbol{\Omega}})$, a list of polynomial lists $\boldsymbol{P} = [\boldsymbol{p}_i]_{i=1}^m$, and equifficient constraints $\boldsymbol{\Lambda} = \{\Lambda_i\}_{i=1}^m$, EPC.Commit computes a commitment c as follows.
First, if for any $p_{i,j} \in \boldsymbol{P}$, $\deg(p_{i,j}) \geq D$, output $\perp$.

Next, compute the commitments to the constrained polynomials. For every list $\boldsymbol{p}_j \in \boldsymbol{P}$ such that Λ_j is a non-trivial constraint, proceed as follows. For notational convenience, set $\boldsymbol{t} := \boldsymbol{p}_j$ and $S := \Lambda_j$. Parse $\boldsymbol{t}$ as $[t_1, \ldots, t_{|S|}]$ and S as $[\mathscr{P}_1, \ldots, \mathscr{P}_{|S|}]$. Denote by t the coefficient vector of the polynomial t_1 in $\mathscr{P}_1$.[13] Construct the 'consistency' commitment of $\boldsymbol{t}$ with respect to S as $c_S := \langle t, \mathsf{ck}_S \rangle \in \mathbb{G}_1$.

Then, compute commitments to the unconstrained polynomials. For every list $\boldsymbol{p}_k \in \boldsymbol{P}$ such that Λ_k is a trivial constraint, proceed as follows. Parse $\boldsymbol{p}_k$ as h and set $V := \Lambda_k$. Commit to h under ck_V via the Pedersen commitment $c_V := \langle h, \mathsf{ck}_V \rangle$. Let W be the first non-trivial constraint in $\boldsymbol{\Omega}$ and let c_W be the commitment to the polynomials in $\boldsymbol{P}$ constrained by W (if there are no such polynomials, i.e., $W \notin \boldsymbol{\Lambda}$, then set $c_W := 0 \cdot G$). Update c_W to $c_W + \sum_{V \in \boldsymbol{\Omega} \setminus \boldsymbol{\Theta}} c_V$.

Finally, output the commitment $c := (c_S)_{S \in \boldsymbol{\Lambda} \cap \boldsymbol{\Theta}} \cup (c_W)$, where $\boldsymbol{\Lambda} \cap \boldsymbol{\Theta}$ denotes the set of non-trivial constraints in $\boldsymbol{\Lambda}$.

[13] Note that if the polynomials in $\boldsymbol{t}$ are equifficient, then t is the coefficient vector for each t_j in the $\mathscr{P}_j$ punctured basis.

Open. On input opener key ok parsed as $\mathsf{ok} := (\mathbb{K}, \langle \mathsf{group} \rangle, \boldsymbol{\Omega}, \boldsymbol{\Sigma}, (\mathsf{ok}_T)_{T \in \boldsymbol{\Omega}})$, a list of polynomial lists $\boldsymbol{P} = [\boldsymbol{p}_i]_{i=1}^m$, a set of equifficient constraints $\boldsymbol{\Lambda} = \{\Lambda_i\}_{i=1}^m$, and an evaluation point u, EPC.Open outputs an evaluation proof π computed as follows.
First, if for any $p_{i,j} \in \boldsymbol{P}$, $\deg(p_{i,j}) \geq D$, output $\perp$.

Next, compute evaluation proofs for the constrained polynomials. For every list $\boldsymbol{p}_j \in \boldsymbol{P}$ such that Λ_j is a non-trivial constraint, proceed as follows. For notational convenience, set $\boldsymbol{t} := \boldsymbol{p}_j$ and $S := \Lambda_j$.

1. Parse $\boldsymbol{t}$ as $[t_1, \ldots, t_{|S|}]$ and ok_S as $(\mathsf{ok}_1, \ldots, \mathsf{ok}_{|S|})$.
2. For each $i \in [|S|]$, compute the unique witness polynomial $w_i(X) = (t_i(X) - t_i(u))/(X - u) \in \mathbb{F}^{<D}[X]$, and commit to it under the opener key ok_i via the Pedersen commitment $\hat{\mathsf{w}}_i := \langle w_i, \mathsf{ok}_i \rangle \in \mathbb{G}_1$.
3. Compute the batched evaluation proof for the polynomials in $\boldsymbol{t}$ as $\mathsf{w}_S := \sum_{i=1}^{|S|} \hat{\mathsf{w}}_i$.

Next, compute evaluation proofs for the unconstrained polynomials. For every list $\boldsymbol{p}_k \in \boldsymbol{P}$ such that Λ_k is a non-trivial constraint, proceed as follows. Parse $\boldsymbol{p}_k$ as h and set $V := \Lambda_k$. Compute the witness polynomial $w(X) = (h(X) - h(u))/(X - u)$, and commit to w under the opener key ok_V via the Pedersen commitment $\mathsf{w}_V := \langle w, \mathsf{ok}_V \rangle \in \mathbb{G}_1$.
Let W be the first non-trivial constraint in $\boldsymbol{\Omega}$ and let w_W be the evaluation proof for the polynomials in $\boldsymbol{P}$ constrained by W (if $W \notin \boldsymbol{\Lambda}$, then set $\mathsf{w}_W := 0$). Set $\mathsf{w}_W := \mathsf{w}_W + \sum_{V \in \Omega \setminus \boldsymbol{\Theta}} \mathsf{w}_V$.

Finally, output the proof $\pi := (\mathsf{w}_S)_{S \in \boldsymbol{\Lambda} \cap \boldsymbol{\Theta}} \cup (\mathsf{w}_W)$, where $\boldsymbol{\Lambda} \cap \boldsymbol{\Theta}$ denotes the set of non-trivial constraints in $\boldsymbol{\Lambda}$.

Verify. On input the verifier key vk parsed as $(\mathbb{K}, \langle \mathsf{group} \rangle, \boldsymbol{\Omega}, (\frac{\alpha_V \cdot G}{\gamma_W})_{V \in \boldsymbol{\Omega} \setminus \boldsymbol{\Theta}}, \{(\frac{\boldsymbol{\alpha}_S \cdot G}{\gamma_S}), (\gamma_S \cdot H), (\gamma_S \tau \cdot H), (\delta_S \cdot G)\}_{S \in \boldsymbol{\Theta}})$, a commitment $c = (c_S)_{S \in \boldsymbol{\Lambda} \cap \boldsymbol{\Theta}} \cup (c_W)$, an evaluation point u, claimed evaluations $\boldsymbol{V} = [\boldsymbol{v}_i]_{i=1}^m$, a set of equifficient constraints $\boldsymbol{\Lambda} = \{\Lambda_i\}_{i=1}^m$, and a proof $\pi = (\mathsf{w}_S)_{S \in \boldsymbol{\Lambda} \cap \boldsymbol{\Theta}} \cup (\mathsf{w}_W)$, EPC.Verify proceeds as follows.
First, check that the constraint Λ_i is contained in the vk; that is, $\Lambda_i \in \boldsymbol{\Omega}$.

Next, let W denote the first non-trivial constraint appearing in $\boldsymbol{\Omega}$. For notational clarity, let $\boldsymbol{v}_T$ denote $\boldsymbol{v}_i$ where i is the index of T in $\boldsymbol{\Lambda}$. If there are no polynomials constrained by W (that is, $W \notin \boldsymbol{\Lambda}$), then set $\boldsymbol{v}_W := \boldsymbol{0}$.
For each constraint $S \in \boldsymbol{\Lambda} \cap \boldsymbol{\Theta} \setminus \{W\}$, check the equality:

$$e(c_S, \delta_S H) = e\left(\frac{\langle \boldsymbol{\alpha}_S, \boldsymbol{v}_S \rangle}{\gamma_S} G, \gamma_S H\right) e(\mathsf{w}_S, \gamma_S \tau H - \gamma_S u \cdot H)$$

Else for $S = W$, check the pairing equality:

$$e(c_W, \delta_W \cdot H) = e\left(\frac{\langle \boldsymbol{\alpha}_W, \boldsymbol{v}_W \rangle}{\gamma_W} \cdot G + \sum_{V \in \boldsymbol{\Lambda} \setminus \boldsymbol{\Theta}} \frac{\langle \boldsymbol{\alpha}_V, \boldsymbol{v}_V \rangle}{\gamma_W}, \gamma_W \cdot H\right) \cdot e(\mathsf{w}_W, \gamma_W \tau \cdot H - \gamma_W u \cdot H)$$