FREPack: Improved SNARK Frontend for Highly Repetitive Computations

Sriram Sridhar[1] and Yinuo Zhang[1]

[1]University of California, Berkeley
srirams@berkeley.edu,yinuo.yz@gmail.com

Abstract

Modern SNARK designs typically follow a frontend-backend paradigm: The frontend compiles a user's program into some equivalent circuit representation, while the backend calls for a SNARK specifically made for proving circuit satisfiability. While these circuits are often defined over small fields, the backend prover always needs to lift the computation to much larger fields to ensure soundness. This gap introduces concrete overheads for ZK applications like zkRollups, where group-based SNARKs are used to provide constant-size proofs for Merkle tree openings.

For a class of *highly repetitive* computations, we propose FREPack, an improved frontend that effectively bridges this gap. The larger the gap between circuit's small field and backend's large field, the more FREPack reduces the circuit size, making it particularly well-suited for group-based backends. Our implementation shows that, for proving ≈ 300 iterations of SHA-256, FREPack improves the performance of Groth16 by $3.6\times$, Nova by $3.8\times$, and Spartan by $5.9\times$.

1 Introduction

In recent years, zero-knowledge Succinct Non-interactive ARguments of Knowledge (zkSNARKs) have gained widespread adoption due to their extremely small proof sizes and efficient verification times. One notable example is zkRollup [rol], which utilizes zkSNARKs to significantly scale the Ethereum blockchain by compressing large numbers of transactions into compact, verifiable proofs.

Modern SNARK Paradigm: Frontend vs. Backend: Modern SNARKs are designed to prove correctness of arbitrary programs through a two-step process:

First, the SNARK **frontend** compiles the program into a specialized circuit representation, where the circuit's satisfiability corresponds to the correct evaluation of the original program. This step is *information-theoretic* and incurs no *computational assumptions*, making it extremely efficient. The efficiency of the frontend is primarily determined by the size of the circuit representation relative to the original program. For instance, in Rank-1 Constraint Systems (R1CS), the circuit size is largely dictated by the number of constraints and variables.

Next, this circuit representation is processed by the SNARK **backend**, which is responsible for proving the *circuit's satisfiability* with a succinct proof. Unlike the frontend, this step involves *cryptographic operations* and relies on *computational assumptions*. For example, in [Gro16], the backend involves multi-scalar exponentiations (MSM) over elliptic curve groups of large order. Broadly speaking, these cryptographic operations can be viewed as arithmetic computations over large finite fields.

Despite extensive research aimed at improving backend efficiency [ZLW+21, GLS+21, XZS22, CBBZ22, DP24], the performance of modern SNARKs appears to be *hitting a bottleneck*, where backend runtimes are constrained to a giant constant factor proportional to the circuit size. Given this limitation, our work shifts focus to the frontend:

Can we theoretically design a more efficient SNARK frontend that minimizes circuit size for certain classes of computations?

From zkRollup to Highly Repetitive Computations: Our investigation along this route is motivated by zkRollup [rol], where the primary challenge lies in generating succinct proofs for large batches of transactions. This often involves proving Merkle tree openings, which require numerous hash function evaluations. For instance, in a Merkle tree with 2^{16} leaves, each path requires 16 hash evaluations, and for a batch of 1000 transactions, the total number of SHA-256 hash evaluations reaches 16,000. A similar pattern occurs in *proof recursion*, where one outer proof system is used to recursively prove the verifier of an inner proof system to further shrink proof size. Since most inner proof systems are hash-based (for performance reasons), this involves proving many Merkle tree openings corresponding to the verifier's query logic in the inner system.

This pattern of repeatedly applying the same subroutine (e.g., a hash function) to different inputs illustrates a broader class of problems known as **highly repetitive computations**. These computations are common across various settings such as data processing pipelines and recursive function calls, where the same operation is performed multiple times in different contexts. We thus broaden our goal to design more efficient frontends that tackle this ubiquitous class of computations. This will benefit all zk-applications where repetitive patterns are prevalent, and backend prover speed remains a limiting factor, more examples to be found in blockchain history data retrieval and zk-virtual machine (zkVM).

Moreover, since on-chain storage is extremely costly due to high gas fees—particularly on networks like Ethereum—it's crucial for zkRollup applications (and similarly in proof recursion, where the final recursed proof almost surely aim to be posted on-chain) to produce *constant-size proofs.* With real-world impact in mind, we also pose the following practical question:

How can we more efficiently generate constant-size proofs for a large number of hash evaluations, such as those in zkRollup and proof recursion?

1.1 Our Contributions

Packing Techniques for Highly Repetitive Computations: We introduce Pack, an information-theoretic method that greatly reduces the size of circuits for repetitive computations over small fields. This is accomplished by redefining the circuit over a larger modulo ring $\mathbb{Z}_q$.

Efficient Emulations for Non-native Modulo Rings: We then introduce FRE, an information-theoretic emulation technique that enables efficiently proving arithmetic relations over rings $\mathbb{Z}_q$ using relations over any fixed prime field $\mathbb{F}_{p^*}$, thus realizing gains from Pack while ensuring compatibility with any existing backend.

Improved Frontend for Highly Repetitive Computations: Combining them into FREPack yields an improved frontend compiler tailored for highly repetitive computations. It is fully compatible with any commit-and-prove SNARK backend operating over large prime fields, especially appealing for group-based backends.

Fast Constant-size Proof Generation for Merkle Openings: We implement FREPack with various group-based backends and our evaluation shows that it greatly reduces the time required to prove repetitive hash computations, such as those found in zkRollup and proof recursion, where constant-size proofs are needed for verifying multiple openings of Merkle trees building on SHA2/SHA3 family.

1.2 Technical Overview

1.2.1 Embedding Overhead

All widely used SNARK frontend compilers transform program logic into circuits defined over *prime fields.* Each wire in the circuit corresponds to an element of the prime field, and constraints represent arithmetic operations (like addition and multiplication) in that field. The prime field is typically chosen based on the size of the program's variables, allowing each variable to be naturally embedded as a field element. For instance, a SHA2-256 circuit can be defined over a small field like $\mathbb{F}_{253}$, which has size $\approx 2^8$.

However, SNARK backends must work with much larger prime fields to ensure cryptographic soundness. For example, when constant-size proofs are required, the backend often operates over an elliptic curve group, corresponding to a prime field $\mathbb{F}_{p^*}$ with size $|p^*| \approx 2^{255}$.

To bridge the gap between the small prime fields used in the frontend and the large fields required by the backend, a common approach is to directly embed the small-field circuits into the larger backend field. For instance, operations originally defined over small fields like $\mathbb{F}_{253}$ are simply embedded into $\mathbb{F}_{p^*}$. Although this naive method works, it introduces a significant **embedding overhead**—the backend prover must perform operations in the large field $\mathbb{F}_{p^*}$ even for constraints that could be efficiently handled in the smaller field. This inefficiency is particularly pronounced in programs with many small-field operations, like cryptographic hash functions (e.g., SHA2, SHA3) or simple bitwise operations – a significant portion of backend field is wasted.

1.2.2 Our Approach

To reduce the embedding overhead, we would like to design better frontends which *information-theoretically* compresses these circuits by trading off some increase in the field size over which the compressed circuit is defined. Since the SNARK backend prover must already work over a large field, compressing the circuit size leads to a direct performance improvement for the prover. We begin to explore this possibility in the space of highly repetitive computations.

1.2.3 Example: Repetitive XOR Computations

In a SHA-2 program, a common operation is the XOR of two ℓ-bit strings: $\mathbf{x} \in \{0,1\}^\ell$ and $\mathbf{y} \in \{0,1\}^\ell$, producing $\mathbf{z} \leftarrow \mathbf{x} \oplus \mathbf{y} \in \{0,1\}^\ell$. This highly repetitive operation is an example of SIMD (Single Instruction, Multiple Data) computation, consisting of ℓ copies of the two-bitwise XOR operation: $\forall i \in [\ell],\ z_i \leftarrow x_i \oplus y_i \in \{0,1\}$.

A straightforward circuit representation for it would consists of ℓ copies of the following constraints:

- $x_i^2 - x_i = 0 \quad y_i^2 - y_i = 0 \quad z_i^2 - z_i = 0 \quad$ (enforcing binary)
- $x_i + y_i - 2x_i \cdot y_i = z_i \quad$ (enforcing $z_i = x_i \oplus y_i$)

This circuit can naturally be defined over the binary field $\mathbb{F}_2$. However, notice that it can also be defined over any prime field $\mathbb{F}_p$ where $p \geq 2$. This is because the first constraints ensure that x_i, y_i, and z_i remain binary. Then out of the 8 possible combinations of (x_i, y_i, z_i), the only way to satisfy the second constraint (over any prime field) is to set $z_i = x_i \oplus y_i$. We refer to this property as **2-satisfiability**, meaning the minimal prime field for this circuit is $\mathbb{F}_2$. In fact, this property is key to enable the aforementioned 'naive' method of embedding each copy of constraints directly into the large field $\mathbb{F}_{p^*}$. The question arises: Can we design a more efficient embedding mechanism that avoids simply repeating ℓ copies of the same constraints?

1.2.4 Pack: Compressing Repetitive Circuits

Our solution is to "pack" the ℓ repetitive copies, each originally defined over a small prime field, into a single circuit defined over a larger algebraic structure. To achieve this, we leverage the Chinese Remainder Theorem (CRT), which states that for any set of ℓ distinct prime numbers $(q_1, \ldots, q_\ell)$, and their product $q = \prod_{i=1}^{\ell} q_i$, the following ring isomorphism holds:

$$\mathbb{Z}_q \cong \mathbb{F}_{q_1} \times \cdots \times \mathbb{F}_{q_\ell}$$

This isomorphism allows us to "pack" circuits defined over the smaller fields $\mathbb{F}_{q_i}$ into a larger circuit defined over the composite ring $\mathbb{Z}_q$.

The process works as follows: for each sub-circuit $i \in [\ell]$, we embed it in a distinct prime field $\mathbb{F}_{q_i}$, such as $q_1 = 2, q_2 = 3$, etc. Since each sub-circuit is **2-satisfiable**, the correctness of the i^{th} sub-circuit is preserved even after embedded.

Now, consider the collection of ℓ sub-circuits as behaving with respect to the direct product $\mathbb{F}_{q_1} \times \cdots \times \mathbb{F}_{q_\ell}$. Using the CRT, we can emulate this behavior in the larger ring $\mathbb{Z}_q$. Thus the packed circuit would still consist of the same constraints, but instead defined over $\mathbb{Z}_q$. Importantly, it only requires one copy, shrinking the circuit size by a factor of ℓ as compared to 'naive embedding'.

In section 3.1, we further observe that any form of highly repetitive computation can be expressed in a similar format to such example of SIMD, when represented as circuits. In section 4.1, we generalize this *information-theoretic* packing technique and introduce our first frontend compiler, Pack, which can efficiently compress circuit representations for any type of highly repetitive computation.

1.2.5 FRE: Handling Non-native Arithmetic Efficiently

A major challenge arises with CRT Packing: almost all SNARK backends operate over a large prime field $\mathbb{F}_{p^*}$, whereas the Packed circuits are defined over the $\mathbb{Z}_q$, where q is a large *composite number*. This means that the packed circuit can no longer be directly embedded into the prime field used by the backend.

This notorious issue is often referred to as non-native arithmetic. Although there are existing solutions for handling non-native arithmetic, most are inefficient, especially when dealing with large composite modulus. These inefficiencies threaten to undo the performance gains achieved through CRT-packing.

As our second contribution, we introduce a technique called *Fast Ring Emulation* (FRE). It is based on an *information-theoretic*, "degree-2 homomorphic" embedding technique that allows the prover to embed elements from $\mathbb{Z}_q$ into (potentially non-unique) elements of $\mathbb{F}_{p^*}$. Using this embedding, the prover can efficiently emulate any degree-2 arithmetic operations over $\mathbb{Z}_q$, including a constant number of additions and scalar multiplications, as well as *one single* multiplication. Crucially, these operations are considered 'SNARK frontend-complete' because the most common circuit representations, such as R1CS (Rank-1 Constraint Systems) and QAP (Quadratic Arithmetic Programs), are built on degree-2 arithmetic constraints.

Illustrative Example: Consider a simple scenario where we need to prove a constraint $a \cdot b = c$, over wire values (a, b, c), and the constraint is defined over non-native ring $\mathbb{Z}_q$.

Since the prover must supply wire values a, b, and c from the native field $\mathbb{F}_{p^*}$, the constraint actually enforces $a \cdot b = c \mod p^*$. Notice that this is problematic even for an honest prover. Just because $a \cdot b = c$ holds over $\mathbb{Z}_q$, it does not imply the same condition over $\mathbb{F}_{p^*}$ due to potential wrap-around. To mitigate this, we allow the prover to supply an additional shift wire value k and turn the constraint into: $a \cdot b = c + k \cdot q$. An honest prover can always find such a shift k so that $a \cdot b = c + k \cdot q$ over the integers, ensuring the relation holds over $\mathbb{F}_{p^*}$. However, a dishonest prover might compute a shift k', allowing $a \cdot b \neq c \mod q$ while still satisfying the constraint $a \cdot b = c + k' \cdot q$ over $\mathbb{F}_{p^*}$.

To counter this attack, we utilize *rational representations* of elements in $\mathbb{F}_{p^*}$. Informally, we say that an element $a \in \mathbb{F}_{p^*}$ is represented as $\frac{a_1}{a_2}$ if $a = \frac{a_1}{a_2} \mod p^*$. Substituting the prover's wire values a, b, c, and k in $\mathbb{F}_{p^*}$ with their rational representations:

$$\frac{a_1}{a_2} \cdot \frac{b_1}{b_2} = \frac{c_1}{c_2} + \frac{k_1}{k_2} \text{ over } \mathbb{F}_{p^*}$$

We claim that if these rationals have small numerators and denominators, the arithmetic holds not only modulo p^* but over the field of rationals as well. To see this, multiply both sides by the least common multiple (LCM) of the denominators gives:

$$a_1 \cdot b_1 \cdot c_2 \cdot k_2 = c_1 \cdot a_2 \cdot b_2 \cdot k_2 + k_1 \cdot a_2 \cdot b_2 \cdot c_2 \cdot q \mod p^*.$$

Since the individual values remain small, no overflow should occur, and the equation holds over the integers. After dividing by the LCM, we recover:

$$\frac{a_1}{a_2} \cdot \frac{b_1}{b_2} = \frac{c_1}{c_2} + \frac{k_1}{k_2} \cdot q.$$

We note that similar observations were made in [CGKR22, GJJZ22] for proving relations over rational numbers. However, in this work we extend the technique further: instead of just viewing $\mathbb{F}_{p^*}$ elements as rationals, we also cast these rational numbers as representatives of $\mathbb{Z}_q$ elements. This corresponds to taking the relation modulo q. For simplicity, assume q is prime[1], making the casted element $a' = \frac{a_1}{a_2} \bmod q$ (similarly for b', c', and k') well-defined. Then our desired relation holds over the casted elements:

$$a' = b' \cdot c' \quad \text{over } \mathbb{Z}_q$$

To summarize, when $\mathbb{F}_{p^*}$ elements are represented as bounded rationals (with small numerators and denominators), they can be viewed as embedded $\mathbb{Z}_q$ elements. This embedding is "somewhat homomorphic," meaning operations in $\mathbb{F}_{p^*}$ project to equivalent operations in $\mathbb{Z}_q$. However, multiplicative homomorphism comes at the expense of increasing the gap requirement between q and p^*. Since our packing efficiency increases with q, we restrict to one multiplication.

Importantly, honest provers can simply use $\mathbb{Z}_q$ elements without overhead. Any $a \in \mathbb{Z}_q$ can be trivially represented as $\frac{a}{1}$, and since in our setting $q \ll p^*$, this representation is a valid bounded rational.

To ensure that the prover's wire values are all bounded rationals, we utilize a recent *information-theoretic* technique underlying Batch Proof-of-Short-Opening (Batch-PoSO). Informally, let $w \in \mathbb{F}_{p^*}^n$ represent all prover's wire values. Batch-PoSO operates by sampling a short random vector $\mathbf{r}$, where each entry is small, and checking if the inner product $\langle \mathbf{r}, w \rangle$ is also small. The intuition is that if this inner product is small with high probability, then by an averaging argument, there must exist two short vectors that differ only at index $i \in [n]$ and both result in small inner products. This allows extraction of each $w[i]$ as a bounded rational.

Building on aforementioned embedding techniques and Batch-PoSO, we introduce *Fast Ring Emulator* (FRE), our second *information-theoretic* frontend compiler which transforms any circuit defined over a non-native modular arithmetic ring into an equivalent circuit over the native prime field.

1.2.6 FREPack: Enhanced Frontend for Repetitive Computations

We combine FRE with the earlier Pack technique, resulting in a unified SNARK frontend called FREPack. This framework integrates seamlessly with any commit-and-prove SNARK backend, and provides substantial efficiency improvements, *particularly for group-based backends*, where the naive embedding overhead is very significant. One subtle issue is that this straightforward integration yields linear verification, nonetheless one can achieve succinct verification through very simple frontend modifications. Additionally, we show that FREPack retains the zero-knowledge property of any backend.

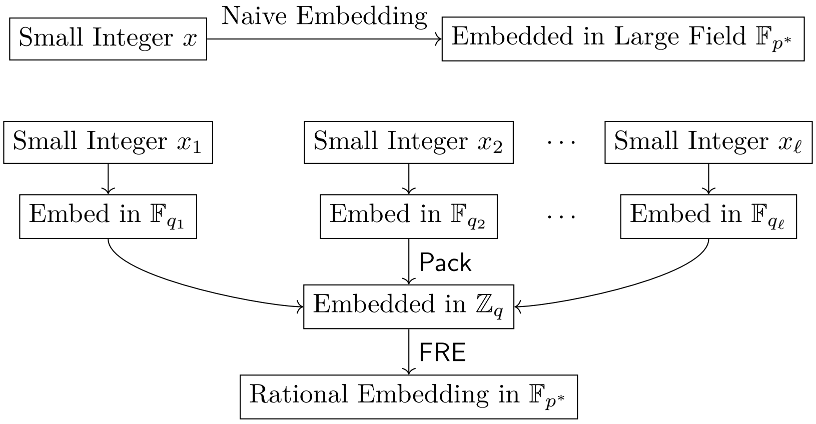

Figure 1: Comparison between Naive Embedding and FREPack Embedding

1.3 Related Works

We divide the related literature into two primary categories: *backend-oriented* and *frontend-oriented* approaches, which differ in their focus on optimizing SNARKs for highly repetitive computations.

[1] As mentioned earlier, q is composite, so inversions are not always well-defined. This complexity is handled later.

1.3.1 Backend-Oriented Approaches:

Backend improvements can be further classified into two distinct subcategories: those that *algorithmically improve the backend* without changing the backend field, and those that *customize the backend field* by moving to smaller fields or binary fields.

Algorithmic Backend Improvements: These works focus on enhancing the backend prover's algorithm while retaining the existing field structure. For instance, the works of [Tha13, WTS$^+$18, XL24] improve various aspects of the prover algorithm originally introduced in [GKR08] for data-parallel (SIMD) computations, where the prover operates over any large prime field. Similarly, for group-based backends, Nova [KST22] and following works [BC23, KS23] introduce efficient folding schemes for incrementally verifiable computation.

Our frontend compiler is fully compatible with all these algorithmic improvements. By combining our frontend optimizations with these algorithmic backend improvements, a "double" speed-up for the prover can be achieved, as demonstrated in Section 7.

Backend Field Customization: This subcategory includes most 'hash' based backends. While those works are not tailored to optimize high repetitive computations, they share the theme of reducing embedding overhead by switching to smaller backend fields. The theoretical line of works [RZR22, BCGL22] builds specific Interactive Oracle Proofs over binary fields. However, these constructions have limitations, such as linear verifier time in [RZR22] and sublinear but still large verifier time in [BCGL22]. Similarly, [AHIV17, KKW18] uses "MPC in the head" to allow backends to work over any finite field, but these protocols result in much larger proofs, making them less practical for real-world use cases.

On the industry side, Plonky3 implements FRI-based backends that use small fields like BabyBear or Mersenne-31, offering faster proving times compared to group-based backends. Similarly, [DP23, DP24] leverage binary extension fields to reduce overhead specifically for binary circuits, though their arithmetic is less friendly for general arithmetic circuits, often relying on expensive lookups to simulate integer arithmetic.

Since these works already reduce the embedding overhead by switching to smaller fields, our FREPack compiler is no longer compatible with those backends. Nonetheless, none of them produce constant-size proofs, making them unsuitable for our target applications. One might ask: why not use recursion with a group-based backend which produces constant-size proof? There are several reasons:

(1) In practice, recursion involves proving many Merkle openings, which is prohibitively expensive. As we demonstrate in appendix section 8.1, unless the number of hashes to be proved exceeds 2^{16}, the recursion cost alone is higher than directly using a group-based backend to prove all hashes.

(2) Proof recursion requires non-black-box use of the hash function, which weakens the security guarantees.

(3) FREPack is specifically designed to improve recursion efficiency, meaning that even when recursion is necessary, FREPack will significantly enhance the process.

1.3.2 Frontend-Oriented Approaches:

Existing frontend optimization approaches are primarily represented by lookup arguments [GW20, ZBK$^+$22]. Lookup arguments reduce the need to check multiple small gate operations in a circuit by performing a single lookup in a large precomputed truth table. For example, the bitwise XOR of two 4-bit strings can be replaced with a lookup in a table containing all 2^8 possible outputs, with each output encoded as a 4-bit value. The nature of lookup is that if the number of lookups becomes comparable to the table size, then good amortization can be achieved. Thus lookup argument only works for circuits with small repetitive structure. In such cases, lookup gates allow for significant reductions in number of gates (circuit constraints) by replacing multiple gates with a single lookup.

Nonetheless, lookup arguments always introduces extra wires (circuit variables) which depend on both the size of the lookup table and the number of lookup gates. For example, in [STW24], the circuits need to remember all access counts in table. In [Hab22], the circuit needs to further maintain LogUp variables for each lookup. Furthermore, if the table is deemed 'unstructured', then the entire lookup table also needs to be materialized in circuit. In some works [GW20, ZBK+22], such cost is even inherent regardless of the table structure.

We observe that for group-based backends, the number of variables plays a more important role than the number of constraints in terms of prover efficiency, due to prover always need to cryptographically commit to all variables. Thus the extra variables introduced in lookup could even decrease overall prover performance, as demonstrated in our SHA2 benchmarking. More details in section 7.

In comparison, FREPack does not require any amortization. Instead, it achieves efficiency by directly reducing size of the circuit, in terms of both the number of gates (constraints) and wires (variables), hence concretely boosts prover efficiency in all scenarios of repetitive computations, including SIMD random circuits where no small repetitive structures to be found. Nonetheless, lookup argument admits other use cases which FREPack does not, particularly in constructing efficient range proofs.

2 Preliminaries

Notation: We denote by λ the security parameter, and let $\mathsf{negl}(\lambda)$ represent a negligible function. That is, for any polynomial $p(\lambda)$, it holds that $\mathsf{negl}(\lambda) < \frac{1}{p(\lambda)}$ for sufficiently large λ. Vectors are denoted by $\mathbf{z}$, with $\mathbf{z}[i]$ referring to the i-th element of $\mathbf{z}$. The inner product between two vectors $\mathbf{z_1}$ and $\mathbf{z_2}$ is denoted by $\langle \mathbf{z_1}, \mathbf{z_2} \rangle$, and the Hadamard (entry-wise) product by $\langle \mathbf{z_1} \circ \mathbf{z_2} \rangle$. For an integer n, the notation $[n]$ represents the set $\{1, 2, \ldots, n\}$. Let $\mathbb{Z}_p$ denote the ring $\mathbb{Z}/p\mathbb{Z}$, i.e., the integers modulo p. When p is a prime, we denote this field by $\mathbb{F}_p$. Throughout the paper we use $\mathbb{F}_{p^*}$ to denote any fixed prime field. The term PPT stands for all *efficient* adversaries, which are algorithms running in probabilistic polynomial time with respect to the security parameter λ. We may refer to these algorithms as *efficient* algorithms throughout the paper.

2.1 Chinese Remainder Theorem (CRT)

Let $(q_1, \ldots, q_n) \in \mathbb{Z}^n$ be a list of n prime numbers, and define $q = \prod_{i=1}^n q_i$. The Chinese Remainder Theorem (CRT) asserts the existence of the following ring isomorphism:

$$\mathbb{Z}_q \cong \mathbb{F}_{q_1} \times \cdots \times \mathbb{F}_{q_n},$$

where the isomorphism is induced by the map $f : \mathbb{Z}_q \to \mathbb{F}_{q_1} \times \cdots \times \mathbb{F}_{q_n}$, defined as: $f(a) = (a \mod q_1, \ldots, a \mod q_n)$.

The inverse map $f^{-1} : \mathbb{F}_{q_1} \times \cdots \times \mathbb{F}_{q_n} \to \mathbb{Z}_q$ is given by:

$$f^{-1}(a_1, \ldots, a_n) = \sum_{i=1}^n a_i \cdot \lambda_i \mod q,$$

where each coefficient λ_i is an integer that satisfies the following properties:

$$\lambda_i \mod q_i = 1 \quad \text{and} \quad \lambda_i \mod q_j = 0 \quad \text{for all } j \neq i.$$

These integers λ_i can be efficiently computed using the following approach. Let: $Q_i = \prod_{j \neq i} q_j$, which is the product of all q_j's except q_i. Then, we define λ_i as: $\lambda_i = Q_i \cdot Q_i^{-1} \mod q_i$, where Q_i^{-1} is the modular inverse of Q_i modulo q_i, i.e., $Q_i \cdot Q_i^{-1} = 1 \mod q_i$.

We also formalize the map f into a set of interfaces, which we refer to as **CRT packing**.

Definition 1 (CRT Packing Scheme)**.** *Let $(q_1, \ldots, q_n)$ be a set of prime numbers, and let $q = \prod_{i=1}^n q_i$. A* ***CRT Packing*** *scheme with respect to this set consists of two algorithms,* (CRT.Pack, CRT.Unpack), *defined as follows:*

- CRT.Pack$(a_1, \ldots, a_n) \to a$*: The packing algorithm takes as input field elements $a_i \in \mathbb{F}_{q_i}$ and packs them into one ring element $a \in \mathbb{Z}_q$.*

- $\mathsf{CRT.Unpack}(a) \to (a_1, \ldots, a_n)$*: The unpacking algorithm takes as input a ring element* $a \in \mathbb{Z}_q$ *and recovers a set of* n *field elements* $(a_1, \ldots, a_n)$ *where* $a_i \in \mathbb{F}_{q_i}$ *for each* $i \in [n]$.

2.2 Vector Commitment Scheme

A vector commitment scheme is a pair of algorithms $(\mathsf{KeyGen}, \mathsf{Commit})$ with the following syntax:

- $\mathsf{KeyGen}(1^\lambda) \to \mathsf{ck}$: The KeyGen algorithm takes as input the security parameter λ, outputs a commitment key ck, and specifies an allowed message space $\mathbb{F}_{p^*}^n$, which is a vector space over some designated prime field.
- $\mathsf{Commit}(\mathsf{ck}, \mathbf{z}) \to c$: The Commit algorithm takes as input a commitment key ck and a vector $\mathbf{z} \in \mathbb{F}_{p^*}^n$, and outputs a commitment c.

We require the following properties to hold:

Succinctness: The size of the commitment c is independent of the length of the vector n.

Binding: Due to the size reduction of commitments, it is possible for different vectors to collide and produce the same commitment. Nevertheless, we require that for all efficient adversaries $\mathcal{A}$, finding such a collision is intractable:

$$\Pr\left[\mathsf{Commit}(\mathsf{ck}, \mathbf{z_1}) = \mathsf{Commit}(\mathsf{ck}, \mathbf{z_2}) \wedge \mathbf{z_1} \neq \mathbf{z_2} : \begin{array}{c} \mathsf{ck} \leftarrow \mathsf{KeyGen}(1^\lambda); \\ (\mathbf{z_1}, \mathbf{z_2}) \leftarrow \mathcal{A}(1^\lambda, \mathsf{ck}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

2.3 Non-interactive Argument of Knowledge

We denote a relation by $\mathcal{R}(\cdot,\cdot)$ and say that a pair consisting of an instance $\mathbb{X}$ and a witness w is in the relation if $\mathcal{R}(\mathbb{X}, w) = 1$. For any relation $\mathcal{R}$, an argument of knowledge for $\mathcal{R}$ consists of the following triple of algorithms $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ with the following interface:

- $\mathsf{Gen}(1^\lambda, \mathcal{R}) \to (\mathsf{pk}, \mathsf{vk})$: The Gen algorithm takes as input the security parameter λ and the description of the relation $\mathcal{R}$, and outputs a public proving key pk and a verification key vk.
- $\mathsf{Prove}(\mathsf{pk}, \mathbb{X}, w) \to \pi$: The Prove algorithm takes as input the proving key pk, an instance $\mathbb{X}$, and an alleged witness w, and outputs a proof π.
- $\mathsf{Verify}(\mathsf{vk}, \mathbb{X}, \pi) \to \{0, 1\}$: The Verify algorithm takes as input the verification key vk, the instance $\mathbb{X}$, and the proof π, and outputs a bit indicating the verification result.

We require the non-interactive argument of knowledge to satisfy the following properties:

Completeness: Completeness requires that for all relations $\mathcal{R}$, we have:

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}, \mathbb{X}, \pi) = 1 : \begin{array}{c} (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{R}); \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pk}, \mathbb{X}, w) \end{array}\right] = 1.$$

Knowledge Soundness: For any efficient adversary $\mathcal{A}$, there exists an efficient extractor $\mathcal{E}$, which has oracle access to $\mathcal{A}$, such that:

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}, \mathbb{X}, \pi) = 1 \wedge \mathcal{R}(\mathbb{X}, w) \neq 1 : \begin{array}{c} \mathcal{R} \leftarrow \mathcal{A}(1^\lambda); \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{R}); \\ (\pi, \mathbb{X}) \leftarrow \mathcal{A}(\mathsf{pk}); \\ w \leftarrow \mathcal{E}^{\mathcal{A}}(\mathsf{vk}, \pi) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Honest Verifier Zero-Knowledge: We say that the non-interactive argument of knowledge is honest-verifier zero-knowledge if there exists a PPT simulator $\mathcal{S}$ such that, for any instance-witness pair $(\mathbb{X}, w)$ in relation $\mathcal{R}$,

$$\{(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{R}), \pi \leftarrow \mathsf{Prove}(\mathsf{pk}, \mathbb{X}, w)\}_{\lambda,\mathbb{X}} \approx \{\mathcal{S}(1^\lambda, \mathbb{X})\}_{\lambda,\mathbb{X}}.$$

2.3.1 Succinct, Non-interactive Argument of Knowledge (SNARK):

A non-interactive argument is succinct if both the proof size $|\pi|$ and the running time of Verify are sublinear in $|w|$, the size of the witness. We are particularly interested in the following special class of SNARKs that build on top of vector commitment schemes:

Definition 2 (Commit-and-Prove SNARKs)**.** *A commit-and-prove SNARK associated with a vector commitment scheme* $(\mathsf{KeyGen}, \mathsf{Commit})$ *and a relation* $\mathcal{R}(\cdot,\cdot)$ *is a SNARK for the following relation* $\mathcal{R}_{\mathsf{ck}}(\cdot,\cdot)$*:*

$$\mathcal{R}_{\mathsf{ck}}((\mathbb{X}, c), w) = 1 \iff \mathcal{R}(\mathbb{X}, w) = 1 \wedge c = \mathsf{Commit}(\mathsf{ck}, w),$$

where $\mathsf{ck} \leftarrow \mathsf{KeyGen}(1^\lambda)$ *is the commitment key.*

Most existing commit-and-prove SNARKs are inherently designed to support proving arithmetic relations over certain prime fields. These fields are sometimes referred to as **native** or **backend** fields and in this paper we always denote them by $\mathbb{F}_{p^*}$. However, the field choices can vary significantly depending on the underlying vector commitment schemes. Here, we provide a brief survey. For more details, readers may consult Chapter 19.3 of [Tha23].

- **Known-order groups**: This category of vector commitment schemes leverages algebraic hardness assumptions in known-order groups. Examples include [KZG10] and [BBB$^+$18]. These schemes are widely adopted in blockchain applications due to their extremely small commitment sizes, which further leads to constant-size proofs. In this work, we are particularly focused on improving backends that utilize known-order groups. The common field choices in this category are scalar fields of the elliptic curve groups, such as BLS12-381, which is a prime field with a 255-bit prime.
- **Collision-resistant hash functions**: This category employs collision-resistant hash functions, such as [COS20] and [ZXZS20]. Here, the field choice is more flexible. As mentioned in section 1.3, for efficiency, smaller prime fields around 32 or 64 bits are often chosen. However, SNARK backends that utilize these smaller fields result in considerably larger proof sizes, limiting their use in blockchain applications. In this work, we do not aim to improve backends using these smaller fields.
- **Unknown-order groups**: The third category utilizes the hardness of unknown-order groups [BFS20, CFKS22, AGL$^+$23, SB23]. These systems are primarily of theoretical interest due to their slower running times, and are beyond the scope of this paper.

2.4 Highly Repetitive Computation

A computation is considered *highly repetitive* if it can be viewed as a fixed sub-computation being applied to multiple pieces of input, which may or may not depend on each other. Examples of such computations include Data Parallel (SIMD) Computation and Incremental Computation.

2.4.1 Data Parallel (SIMD) Computation

Data parallel computation, or Same Instruction Multiple Data (SIMD), is a common form of highly repetitive computation where the same sub-computation is applied to multiple independent inputs. This type of computation is ubiquitous in many real-world applications.

As a concrete example, consider the SIMD computation G^{SIMD}, where the sub-computation G is repeated ℓ times on ℓ different independent inputs $(x_1, \ldots, x_\ell)$. That is:

$$G^{\mathsf{SIMD}}(x_1, \ldots, x_\ell) = (G(x_1), \ldots, G(x_\ell)).$$

2.4.2 Incremental Computation

Incremental computation typically involves a sub-computation being applied iteratively to a sequence of dependent inputs, capturing most recursive functions and while loops in a program.

As an example, consider the iterative computation G^{IC}, where the sub-computation G is applied ℓ times in sequence, with each output serving as the input for the next iteration. That is:

$$G^{\mathsf{IC}}(x) = \underbrace{G(G(\ldots G(x)))}_{\ell \text{ times}}.$$

3 Existentially Quantified Circuits (EQC)

Typically, a SNARK frontend compiler transforms a user's program into a suitable circuit representation. In this work, we adopt an abstract view of this circuit representation, formalized as an *Existentially Quantified Circuit* (EQC) [OBW22].

Definition 3 (Existentially Quantified Circuits (EQC)). *An EQC consists of a set of wires that take values from a specified domain (such as the prime field $\mathbb{F}_p$) and constraints that express relationships among the wire values (e.g., the constraint $x \cdot y = z$). There are two types of wire values: explicit input/output values, which are assigned to input/output wires at the start of execution, and existentially quantified wire values, which can take any value consistent with the input values and the constraints. We denote an EQC by the tuple $(\mathbb{F}, C, io, w)$, representing the domain, the set of constraints, the input/output values, and the existentially quantified wire values.*

*We say an EQC $(\mathbb{F}, C, io)$ is **satisfiable** if there exists a set of existentially quantified wire values $w \in \mathbb{F}$ such that (io, w) together satisfy all the constraints in C. When the domain $\mathbb{F}$ is clear from the context, we define the induced relation $\mathcal{R}^C$ such that an instance-witness pair $(io, w) \in \mathcal{R}^C$ if (io, w) satisfy all the constraints described by C.*

3.1 Reducing Highly Repetitive Computations to SIMD EQCs

In the context of EQCs, we observe that all highly repetitive computations can in fact be reduced to a specialized class of circuits:

Definition 4 (SIMD EQC). *For any number ℓ, a SIMD EQC $\{(\mathbb{F}, C, io_i)\}_{i\in[\ell]}$ consists of ℓ copies of the same circuit constraints C, each with respective input/output values $(io_1, \ldots, io_\ell)$. It is satisfiable if all ℓ internal copies are satisfiable. Let $\mathcal{R}^{C,\mathsf{SIMD}}$ denote the SIMD relation w.r.t. C. We say that the instances and existential wires $\left(\{io_i\}_{i\in[N]}, \{w_i\}_{i\in[N]}\right)$ are in the relation $\mathcal{R}^{C,\mathsf{SIMD}}$ if, for each $i \in [N]$, the existential wires w_i satisfy the EQC instance $(\mathbb{F}_{p^*}, C, io_i)$.*

It should be obvious that any SIMD computation can be naturally expressed as its SIMD EQC. For all other types, let's start by considering the incremental computation $G^{\mathsf{IC}}(x) = \underbrace{G(G(\ldots G(x)))}_{\ell \text{ times}}$. We can convert this into a SIMD by leveraging the non-determinism of EQCs and introducing existential wires to verify the consistency of transitions between iterations.

Let x_i be the input to the i^{th} iteration of the sub-computation G, and let y_i be the corresponding output such that $G(x_i) = y_i$. This transformation yields a SIMD computation $G^{\mathsf{SIMD}}(x_1, \ldots, x_\ell) \to (y_1, \ldots, y_\ell)$, where $y_i = G(x_i)$. For this sub-computation, we derive its equivalent SIMD EQC. Nonetheless, given the iterative structure, additional consistency constraints must be enforced to ensure correctness: $x_i = y_{i-1}$ for all $i \in [\ell]$.

This methodology extends to other highly repetitive computations by adding suitable consistency constraints, ensuring the proper relationships between intermediate wire values are maintained. In some cases, the entire computation may not be highly repetitive, but it could contain a large repetitive subcomponent. In such scenarios, we isolate the repetitive sub-component from the rest of the computation while tracking the set $S = \{w_i\}$ of shared wires between the two components. Eventually, we add constraints to all shared wires $w_i \in S$ to ensure consistency between the two components.

3.2 $p_{\min}$-Satisfiability of EQCs over Prime Fields

Nearly all currently deployed SNARK frontends, such as circom, output a family of EQCs defined over a specific prime field $\mathbb{F}_p$. This implies that all wire values are elements of $\mathbb{F}_p$, and the constraints correspond to arithmetic operations (addition, subtraction, multiplication) over $\mathbb{F}_p$.

A key observation in this work is that these frontends generate EQCs that exhibit an interesting property: there exists a minimal prime field $\mathbb{F}_{p_{\min}}$ such that for any prime $p' \geq p_{\min}$, the EQC is satisfiable over $\mathbb{F}_{p'}$ if and only if it is satisfiable over $\mathbb{F}_{p_{\min}}$. Intuitively, the program's logic can be sufficiently expressed in this minimal field, yet any larger prime field can still embed it. For example, consider an EQC with a single constraint $x(x-1) = 0$, which enforces x to be binary. This EQC is $p_{\min}$-satisfiable with $p_{\min} = 2$.

Definition 5 ($p_{\min}$-Satisfiability)**.** *Let $(\mathbb{F}_p, C, io)$ be an EQC defined over some prime field $\mathbb{F}_p$ with input/output values $io \in \mathbb{F}_p$. We say it is $p_{\min}$**-satisfiable** with respect to a minimal prime number $p_{\min}$ if:*

For any prime $p' \geq p_{\min}$, there exists some $io' \in \mathbb{F}_{p'}$,[2] such that $(\mathbb{F}_{p'}, C, io')$ is satisfiable if and only if $(\mathbb{F}_p, C, io)$ is satisfiable.

3.3 The Gap Between $\mathbb{F}_{p_{\min}}$ and the Backend Field

We observe that for most computer programs involving only bit or small integer operations, their logic can be expressed in small minimal prime fields. In particular, for AES, SHA2, and SHA3, the frontend compilers [OBW22] [KPS18] [cir] output EQCs with $p_{\min} < 2^8$.

However, to ensure SNARK soundness, $\mathbb{F}_{p_{\min}}$ must be embedded in the backend field $\mathbb{F}_{p^*}$ is often significantly larger—by orders of magnitude.

This substantial discrepancy in field sizes leads to considerable overhead for the prover. Conceptually, much of the prover's computational effort in the larger backend field is redundant, as a significant portion of the work doesn't contribute directly to the verification of the original program logic.

4 First Technique: CRT Packing

We now partially bridge the gap between $\mathbb{F}_{p_{\min}}$ and the backend field in the context of highly repetitive computations using an **information-theoretic** packing technique based on the CRT isomorphism.

Informally, this isomorphism allows us to emulate the arithmetic behaviors of multiple small, distinct prime fields using a larger ring of modulo arithmetic. This provides a compression mechanism that absorbs more computations, thereby narrowing the field size gap. To illustrate this concept more clearly, we begin by focusing on any SIMD EQC $\{(\mathbb{F}_p, C, \mathrm{io}_i)\}_{i\in[\ell]}$. As mentioned earlier, the circuits of all repetitive computations can be reduced to this very essential form.

4.1 Pack: Efficient Frontend Compiler for SIMD EQC

Let $p_{\min}$ be the prime such that the above SIMD EQC is $p_{\min}$-satisfiable. We design the following frontend compiler, denoted as Pack (Fig. 2) that utilizes the CRT isomorphism to compress its ℓ internal copies.

$\underline{\mathsf{Pack}(\{(\mathbb{F}_p, C, \mathrm{io}_i)\}_{i\in[\ell]})}$

1. *Choose CRT Basis:* Choose ℓ smallest distinct prime numbers $(q_1, \ldots, q_\ell)$ such that each $q_i \geq p_{\min}$. Let $q = \prod_{i=1}^{\ell} q_i$.
2. *Pack the input/output wire values:* For each $i \in [\ell]$, let io'_i be defined as in Definition 5. Then apply $\mathsf{CRT.Pack}(\mathrm{io}'_1, \ldots, \mathrm{io}'_\ell) \to \mathrm{io}'$ with respect to the above basis.
3. Define the final packed EQC instance to be $(\mathbb{Z}_q, C, \mathrm{io}')$.

Figure 2: Pack: A Frontend Compiler for SIMD $p_{\min}$-satisfiable EQCs.

Remark 1. *When the CRT basis is clear from the given context, we denote by $io \leftarrow \mathsf{Pack}(\{io_i\}_{i\in[\ell]})$ the packed input/output values, and similarly for existential wire values.*

Theorem 1 (Completeness and Soundness of Pack)**.** *Assuming that the SIMD EQC instance $\{(\mathbb{F}_p, C, io_i)\}_{i\in[\ell]}$ is $p_{\min}$-satisfiable, let the packed EQC instance $(\mathbb{Z}_q, C, io')$ be defined as in figure 2. Then, $(\mathbb{Z}_q, C, io')$ is satisfiable if and only if $\{(\mathbb{F}_p, C, io_i)\}_{i\in[\ell]}$ is satisfiable.*

Proof. The claim follows directly from the CRT ring isomorphism and the definition of $p_{\min}$-satisfiability of EQC. □

[2] In almost all applications, we have $\mathrm{io} = \mathrm{io}'$ as integers.

The Packing Factor: Attentive readers may notice that as the number of copies increases, the size of the ring $\mathbb{Z}_q$ also grows, gradually closing the gap between $\mathbb{Z}_q$ and the backend field $\mathbb{F}_{p^*}$. Eventually, this gap narrows to a point where further Packing is no longer feasible. To address this, we first define a maximum limit on the number of copies that can be packed together, which we refer to as the *packing factor*, denoted by ℓ. For any SIMD EQC consisting of N copies, we split them into N/ℓ batches and Pack each batch individually.

4.2 Ensuring Backend Compatibility

While the frontend compiler Pack provides a powerful mechanism to compress highly repetitive circuits, it introduces an incompatibility issue: Pack produces EQCs defined over the modular arithmetic ring $\mathbb{Z}_q$, where q is a composite number, hence can no longer be embedded in the desirable backend field $\mathbb{F}_{p^*}$.

This incompatibility falls under the broader challenge of proving non-native arithmetic, which is a significant hurdle in SNARK design. Existing solutions to this problem, while functional, are generally inefficient. Some approaches, such as [KPS18], rely on bit decomposition, which introduces a large number of constraints, significantly reducing efficiency. Other methods, such as those involving special ring encodings [GNSV21], sacrifice desirable SNARK features like public verifiability and still impose substantial overhead on the backend. The bottom line is that utilizing any of these solutions would negate the frontend efficiency gains achieved through CRT packing.

5 Second Technique: Fast Ring Emulation

Our second major contribution is the development of an efficient frontend technique to handle the non-native arithmetic present in this specific setting, which we call **Fast Ring Emulation (FRE)**. The core of this technique is an *information-theoretic*, degree-2 homomorphic embedding scheme. This scheme enables the prover to embed elements from $\mathbb{Z}_q$ into (potentially non-unique) elements of $\mathbb{F}_{p^*}$. Using these embedded elements, the prover can efficiently emulate any degree-2 arithmetic operations over $\mathbb{Z}_q$, including:

- Any constant number of additions or scalar multiplications, and
- One single multiplication.

These operations are crucial because they are considered *SNARK-complete*, meaning that they are sufficient to express the EQCs typically used in SNARKs. For instance, [Set20, BCR$^+$19, KST22] utilize Rank-1 Constraint Systems (R1CS), while [GGPR13, Gro16] target Quadratic Arithmetic Programs (QAP). Moreover, in practical applications, such as the Cairo zkVM project [GPR21], specifically degree-2 Algebraic Intermediate Representations (AIR) are adopted.

To streamline our technical discussions, we first focus on characterizing degree-2 EQCs through the following complete arithmetic constraint:

Definition 6 (Degree-2 Complete Arithmetic Constraint)**.** *Without loss of generality, for any degree-2 EQC $(\mathbb{F}, C, io)$ with existentially quantified wire values w, we assume that each arithmetic constraint in C follows the pattern:*

$$\left(\sum_{a_i \in S} a_i\right) \cdot b = c, \tag{1}$$

where $S \subseteq io \cup w$ is a subset whose wire values sum to the left input of the multiplication gate, and $b, c \in io \cup w$ are the right input and output values, respectively. We also assume, without loss of generality, that all gate constraints have constant fan-in. That is, there exists a fixed constant $\mathbf{c}$ such that $|S| \leq \mathbf{c}$.

5.1 Rational Representation

Our goal is to handle the aforementioned constraints over a non-native ring $\mathbb{Z}_q$. In reality, the prover can only supply existential wire values from the native field $\mathbb{F}_{p^*}$. So we must view the values a_i, b, and c as elements of $\mathbb{F}_{p^*}$. Despite this, we aim to jump outside reality and reimagine these values and convert them into a suitable embedding of $\mathbb{Z}_q$ elements.

To achieve this, we utilize a less common but useful representation of $\mathbb{F}_{p^*}$ elements, known as a **rational representation**. Informally, we say that an element $a \in \mathbb{F}_{p^*}$ can be represented by a rational expression $\frac{a_1}{a_2}$ if it holds that $a = \frac{a_1}{a_2} \mod p^*$. For simplicity, let's assume that p^* is a prime, ensuring that this relation is always well-defined. We defer further details on rational representatives to Section 5.5.

5.2 Reformulating Constraints Over the Field of Rationals

Substituting each $\mathbb{F}_{p^*}$ element with its rational representation in 1, we obtain:

$$\sum_{a_i \in S} \frac{a_{i,1}}{a_{i,2}} \cdot \frac{b_1}{b_2} = \frac{c_1}{c_2} \mod p^*. \tag{2}$$

Observe the following: If all these rational representatives have numerators and denominators much smaller than p^* (e.g., $a_{i,1}, a_{i,2}, \ldots, b_1, \cdots \ll p^*$), then Equation 2 indeed holds over the field of rational numbers, not just modulo p^*.

To see this, let L be the least common multiple (LCM) of $\{a_{i,2}\}_{a_i \in S}$. Multiplying both sides of Equation 2 by $L \cdot b_2 \cdot c_2$, we obtain:

$$\sum_{a_i \in S} \frac{L}{a_{i,2}} \cdot (a_{i,1} b_1 c_2) = L \cdot b_2 c_1 \mod p^*. \tag{3}$$

Since each individual variable is assumed to be much smaller than p^*, and both sides are products over a constant number of variables, the resulting values remain small integers less than p^*. Therefore, due to the absence of wrap-around, Equation 3 actually holds over the integers. Dividing both sides by $L \cdot b_2 \cdot c_2$, we see that Equation 2 now holds over the rationals.

In other words, whenever the prover uses rational representatives with small numerators and denominators, they can completely avoid dependency on the field $\mathbb{F}_{p^*}$ by reformulating the same arithmetic relation over the field of rationals. This rational embedding technique is also employed in [CGKR22, GJJZ22] for the purpose of proving arithmetic relations over rational numbers and developing range proofs.

5.3 Casting Rationals Back to $\mathbb{Z}_q$ Elements

In this work, we extend the rational embedding technique one step further: On top of viewing $\mathbb{F}_{p^*}$ elements as rationals, we once again cast these rational numbers as elements in a finite domain. More specifically, we reimagine these rational numbers as rational representatives of $\mathbb{Z}_q$ elements. This is equivalent to taking the equation over modulo q. For simplicity, let's assume that q is a prime, so that any casted element $\tilde{a} = \frac{a_1}{a_2} \mod q$ is always well-defined. We ultimately obtain the following equation:

$$\left(\sum_{\tilde{a_i} \in S} \tilde{a_i} \right) \cdot \tilde{b} = \tilde{c} \mod q. \tag{4}$$

By using rational representations as intermediaries, the prover can prove $\mathbb{Z}_q$ modulo arithmetic relations, even though the witness values are actually $\mathbb{F}_{p^*}$ elements. In summary, when the prover uses bounded rational representatives (with small numerators and denominators), these elements can be viewed as embedded $\mathbb{Z}_q$ elements. This embedding is "somewhat homomorphic," meaning that arithmetic operations in $\mathbb{F}_{p^*}$ naturally project to operations in $\mathbb{Z}_q$.

Importantly, honest provers can safely use $\mathbb{Z}_q$ elements for proving arithmetic relations. For instance, any element $a \in \mathbb{Z}_q$ can be written as its rational representative $\frac{a}{1}$. Since $a < q \ll p^*$, this representation is always valid, ensuring no overhead for honest provers.

5.4 Enforcing Bounded Rationals with Batch-PoSO:

To enforce bounded rationals, we leverage a recent technique called Batch Proof-of-Short-Opening (Batch-PoSO) [CGKR22, GJJZ22]. We refer readers back to section 1.2.5 for its high-level overview. Its details are deferred to section 5.6.

Enforcing Well-defined Rationals: Since CRT packing requires q to be a composite number, rational representatives in $\mathbb{Z}_q$ may not always be well-defined due to the absence of an inverse. To mitigate this, we observe that if a rational's denominator is sufficiently bounded, i.e., smaller than the smallest divisor of q, the rational remains well-defined. Batch-PoSO can also be used to enforce this condition. More details are deferred to lemma 1.

5.5 Related Concepts of Rational Representatives

We adopt the following definition of rational representatives from [CKLR21, CGKR22]:

Definition 7 (Rational Representative)**.** *Let $\mathbb{Q}$ denote the set of rational numbers, where the numerator and denominator are coprime:*

$$\mathbb{Q} = \left\{ \frac{n}{d} \mid n, d \in \mathbb{Z}, \gcd(n, d) = 1 \right\}.$$

For any element $x \in \mathbb{F}_p$, we say x is represented by the rational $\frac{n}{d} \in \mathbb{Q}$ if it holds that $x = n \cdot d^{-1} \mod p$.

Note that each element $x \in \mathbb{F}_p$ can have multiple rational representatives. This concept can be generalized to any ring $\mathbb{Z}_q$ where q is not necessarily prime. In this case, we restrict ourselves to the set of rational numbers whose denominators are invertible modulo q.

Definition 8 (q-Invertible Rational Representative)**.** *Let $\mathbb{Q}_q$ be the set of rationals where the denominator is coprime to q:*

$$\mathbb{Q}_q = \left\{ \frac{n}{d} \mid n, d \in \mathbb{Z}, \gcd(n, d) = 1, \gcd(q, d) = 1 \right\}.$$

Definition 9 (Bounded Rational Representative)**.** *The set of bounded rationals $\mathbb{Q}_{N,D} \subseteq \mathbb{Q}$ consists of all rationals whose numerators are bounded by N and denominators are bounded by D:*

$$\mathbb{Q}_{N,D} = \left\{ \frac{n}{d} \mid |n| \leq N, |d| \leq D \right\} \subseteq \mathbb{Q}.$$

We observe the following relationship between the set $\mathbb{Q}_q$ and $\mathbb{Q}_{N,D}$:

Lemma 1 (Criterion for q-Invertibility)**.** *Let $q_{\min}^{\mathsf{divisor}}$ be the smallest divisor of q. If $D < q_{\min}^{\mathsf{divisor}}$, then all rationals in $\mathbb{Q}_{N,D}$ are also q-invertible. That is, $\mathbb{Q}_{N,D} \subseteq \mathbb{Q}_q$ when $D < q_{\min}^{\mathsf{divisor}}$.*

Proof. Since any denominator smaller than $q_{\min}^{\mathsf{divisor}}$ must be coprime to q, the corresponding rational number is q-invertible. In all our applications, we will set q such that $q_{\min}^{\mathsf{divisor}}$ is relatively large (e.g., $q_{\min}^{\mathsf{divisor}} \approx 2^8$). □

5.6 Batch Proof of Short Opening (Batch-PoSO)

We slightly extend our notation as follows: for any $x \in \mathbb{F}_{p^*}$, we denote $x \in \mathbb{Q}_{N,D}$ if x has a rational representative in $\mathbb{Q}_{N,D}$. In our application, we aim to enforce that the prover's wire values w are contained within $\mathbb{Q}_{N,D}$. This issue has been addressed in [CKLR21] and later works [CGKR22, GJJZ22], which propose an efficient commit-and-prove protocol (definition 2) called Batch Proof of Short Opening (Batch-PoSO). The name reflects its goal: proving that a 'short' (i.e., bounded) rational representative exists for each entry in a committed vector[3], hence the term 'short opening.'

However, proving this exact condition efficiently is infeasible. Instead, Batch-PoSO works by relaxing the goal slightly, introducing a gap in the relation it seeks to prove.

Definition 10 (Gap Relation ($\mathcal{R}_{R,1}, \mathcal{R}_{N,D}$))**.** *A vector $w \in \mathbb{F}_{p^*}^n$ is said to belong to the relation $\mathcal{R}_{R,1}$ if $w \subseteq \mathbb{Q}_{R,1}$. It belongs to the relation $\mathcal{R}_{N,D}$ if $w \subseteq \mathbb{Q}_{N,D}$.*

In this gap relation setting, the verifier must always accept all instances in $\mathcal{R}_{R,1}$ and reject, with high probability, all instances not in $\mathcal{R}_{N,D}$. When emulating the ring $\mathbb{Z}_q$, we set $R = q \ll N$, ensuring that all elements of the emulated ring fall into the accepting relation, so the honest prover always succeeds in Batch-PoSO. This gap is a crucial component leveraged by our Fast Ring Emulation technique, as it relaxes the tightness of relation, allowing for more efficient proof technique.

[3]In our final application, the wire values w are also represented as a committed vector.

Batch-PoSO Overview: The core of Batch-PoSO is an *information-theoretic* property testing framework outlined in fig. 3. This information-theoretic framework can be easily compiled into a commit-and-prove system by having the prover first commit to w, after which the verifier samples the random vector $\mathbf{r}$. We defer the details of this compilation to the next section.

Property Testing Framework for Gap Relation $(\mathcal{R}_{R,1}, \mathcal{R}_{N,D})$

- **Protocol Description:**
  1. Sample a short random vector $\mathbf{r} \in \mathbb{F}_{p^*}^n$ such that each entry $\mathbf{r}[i]$ is small (i.e. $\forall i \in [n], \mathbf{r}[i] \in [0, D)$).
  2. Compute $v :=< w \cdot \mathbf{r} >$ over $\mathbb{F}_{p^*}$, and accept if and only if $v \in [N]$.

Figure 3: An Information-Theoretic Property Testing Protocol.

5.6.1 Security proof for Batch-PoSO

We prove that the property testing protocol described in figure 3 satisfies completeness and soundness.

Theorem 2 (Completeness)**.** *Whenever $N \geq R \cdot D \cdot n$, the protocol satisfies completeness.*

Proof. It is easy to see that if for each $i \in [n]$, $w[i] \in [R]$, then $v := (< w \cdot \mathbf{r} >) \quad < (R \cdot D \cdot n) \leq N$. Thus $v \in [N]$ and we will always accept. □

Theorem 3 (Soundness)**.** *The protocol admits soundness error of at most $1/D$.*

We prove soundness by utilizing the following extraction lemma:

Lemma 2 (Extraction Lemma)**.** *For any fixed constant N, and for any $w \in \mathbb{F}_{p^*}^n$, if*

$$\Pr\left[r_1, r_2, \ldots, r_n \leftarrow [0, D) : \sum_{i=1}^{n} r_i \cdot w[i] < N \mod p^*\right] > 1/D,$$

then for each i, there exists two integers $w_{i,1} \in [-N, N]$ and $w_{i,2} \in [1, D]$ such that $w[i] = \frac{w_{i,1}}{w_{i,2}} \mod p^*$.

Proof. The proof relies on probabilistic method. More specifically, since we have:

$$\Pr_{r_1, r_2, \ldots, r_n \leftarrow [0,D)}\left[\sum_{i=1}^{n} r_i \cdot w[i] < N \mod p^*\right] > 1/D,$$

by averaging argument, for each $i \in [n]$, there must exist some fixed $(r_1^*, r_2^*, \ldots, r_{i-1}^*, r_{i+1}^*, \ldots, r_n^*)$ such that

$$\Pr_{r_i \leftarrow [0,D)}\left[\sum_{j \neq i}^{n} r_j^* \cdot w[j] + r_i \cdot w[i] < N \mod p^*\right] > 1/D.$$

Since there are only D choices of r_i, there must exist two $r_{i,1}, r_{i,2} \in [0, D), (r_{i,1} > r_{i,2})$ such that

$$\sum_{j \neq i}^{n} r_j^* \cdot w[j] + r_{i,1} \cdot w[i] \in [N] \bigwedge \sum_{j \neq i}^{n} r_j^* \cdot w[j] + r_{i,2} \cdot w[i] \in [N].$$

Now we set $w_{i,2} := r_{i,1} - r_{i,2} \in [1, D)$, and set

$$w_{i,1} := \left(\sum_{j \neq i}^{n} r_j^* \cdot w[j] + r_{i,1} \cdot w[i]\right) - \left(\sum_{j \neq i}^{n} r_j^* \cdot w[j] + r_{i,2} \cdot w[i]\right)$$

as the difference between previous two sums. Notice that $w_{i,1} \in [-N, N]$. Finally, observe that $w[i] = \frac{w_{i,1}}{w_{i,2}}$ mod p^*, where $w_{i,1} \in [-N, N], w_{i,2} \in [1, D]$ as desired. □

5.7 FRE: Bringing Fast Ring Emulation into the Frontend

In this section, we formalize how to use Fast Ring Emulation (FRE) as a frontend technique to address challenges related to non-native modulo arithmetic. Specifically, let $(\mathbb{Z}_q, C, \mathrm{io})$ be an EQC instance defined over a non-native ring $\mathbb{Z}_q$. Our goal is to design a frontend compiler that compiles such an EQC into an equivalent instance defined over the native field $\mathbb{F}_{p^*}$, while maintaining the same satisfiability condition as the original $(\mathbb{Z}_q, C, \mathrm{io})$ with overwhelming probability. We name this frontend compiler the Fast Ring Emulator (FRE).

5.7.1 Fitting Stage

In the first stage, FRE.Fit, we transform each wiring constraint in the non-native EQC instance $(\mathbb{Z}_q, C, \mathrm{io})$ into a corresponding constraint over $\mathbb{F}_{p^*}$ by introducing shift wire values to account for the modulus differences between $\mathbb{Z}_q$ and $\mathbb{F}_{p^*}$. The goal of this transformation is to ensure that if $(\mathbb{Z}_q, C, \mathrm{io})$ is satisfiable, then $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}})$ will also be satisfiable. However, the reverse implication (satisfiability of the transformed instance implying satisfiability of the original instance) does not necessarily hold.

After this stage, the prover must also update the wire values from w to w_{Fit}, incorporating the shift values. The full description of FRE.Fit is provided in figure 4.

Claim 1. *For any satisfiable EQC instance $(\mathbb{Z}_q, C, io)$ with existential wire values w, there exists a set of shift wire values $\{k^j\} \in \mathbb{Z}_q$ such that $w \cup \{k^j\} \in \mathbb{Z}_q$ makes the fitted EQC $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, io_{\mathsf{Fit}})$ satisfiable.*

Proof. Since each constraint j is degree-2, there always exists $k^j \in \mathbb{Z}_q$ such that $\left(\sum_{a_i^j \in S^j} a_i^j\right) \cdot b^j = c^j + k^j \cdot q$ holds over the integers, and thus it also holds over $\mathbb{F}_{p^*}$. Therefore, the fitted EQC $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}})$ is satisfiable. □

5.7.2 Emulating Stage

The second stage, FRE.Emulate, ensures that all prover's wire values w_{Fit} are valid embeddings of $\mathbb{Z}_q$ elements. As discussed in section 5.5, this is equivalent to enforcing that all wire values are bounded rational representatives. This is achieved by arithmetizing the *information-theoretic property testing framework* that underlies Batch-PoSO.

To achieve this, FRE.Emulate translates the property testing framework into an equivalent EQC instance $(\mathbb{F}_{p^*}, C_{\mathsf{PoSO}}, \mathrm{io}_{\mathsf{PoSO}})$, such that the instance is satisfiable over $\mathbb{F}_{p^*}$ with respect to the existential wires w_{Fit} and $\mathrm{io}_{\mathsf{PoSO}}$ if and only if the property testing framework accepts w_{Fit} over randomness $\mathrm{io}_{\mathsf{PoSO}}$. This ensures w_{Fit} are valid embeddings with high probability, where the probability is taken over the random input.

The final EQC instance after this stage is the concatenation of $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}})$ and $(\mathbb{F}_{p^*}, C_{\mathsf{PoSO}}, \mathrm{io}_{\mathsf{PoSO}})$. This ensures that the overall instance is satisfiable if and only if the original instance $(\mathbb{Z}_q, C, \mathrm{io})$ is satisfied, with all but the error probability introduced in property testing. By lemma 3, since a single invocation incurs a statistical soundness error of $1/D$, the final instance $(\mathbb{F}_{p^*}, C_{\mathsf{PoSO}}, \mathrm{io}_{\mathsf{PoSO}})$ will repeat property testing $\lambda/\log(D)$ times to achieve negligible soundness error. The full description of FRE.Emulate is provided in figure 5.

Remark 2 (Existential Wiring Consistency)**.** *The final EQC instance is obtained by concatenating the EQC constraints $(C_{\mathsf{Fit}} \cup C_{\mathsf{PoSO}})$, and the inputs $(io_{\mathsf{Fit}} \cup io_{\mathsf{PoSO}})$. Importantly, the concatenation must respect the same set of existential wires across both circuits. This means $w_{\mathsf{PoSO}} = w_{\mathsf{Fit}}$. In practice, this requirement is naturally achieved by using commit-and-prove SNARK backends, where the commitment to the existential wires is shared across both EQC instances.*

Theorem 4 (Emulation Completeness)**.** *If the original EQC instance $(\mathbb{Z}_q, C, io)$ is satisfiable, then the concatenated instance $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}} \cup C_{\mathsf{PoSO}}, io_{\mathsf{Fit}} \cup io_{\mathsf{PoSO}})$ must always be satisfiable.*

Proof. Let $w \in \mathbb{Z}_q^{|w|}$ be the existential wires that satisfy $(\mathbb{Z}_q, C, \mathrm{io})$. By theorem 1, there exists a set of augmented wires $w_{\mathsf{Fit}} \in \mathbb{Z}_q^{|w_{\mathsf{Fit}}|}$ that make $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}})$ satisfiable. Since $w_{\mathsf{Fit}} \subseteq \mathbb{Q}_{R,1}$, it will pass the property testing framework of Batch-PoSO (completeness in claim 2). Since $(\mathbb{F}_{p^*}, C_{\mathsf{PoSO}}, \mathrm{io}_{\mathsf{PoSO}})$ is

$\underline{\mathsf{FRE.Fit}(\mathbb{Z}_q, C, \mathrm{io})}$

1. Initialize $C_{\mathsf{Fit}} = \emptyset$. For each wire constraint $j \in C$:
   - Parse it in the form of a degree-2 arithmetic constraint:
$$\sum_{a_i^j \in S^j} a_i^j \cdot b^j = c^j \mod q.$$
   - Add a shift variable k^j as an existential wire value, and append the following constraint to C_{Fit}:
$$\sum_{a_i^j \in S^j} a_i^j \cdot b^j = c^j + k^j \cdot q \mod p^*.$$
2. Output the updated instance $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}} = \mathrm{io})$.
   After this stage, the existential wire values are updated to $w_{\mathsf{Fit}} = w \cup \{k^j\}_{j\in[1,|C|]}$.

Figure 4: Fitting Stage of FRE

$\underline{\mathsf{FRE.Emulate}(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}})}$

1. Let w_{Fit} be the wire values of the EQC instance $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}})$.
2. *Set* PoSO *Parameters:*
   Let $q_{\min}^{\mathsf{divisor}}$ be the smallest divisor of q. Define the gap relation $(\mathcal{R}_{R,1}, \mathcal{R}_{N,D})$, where $R = q$, $D = q_{\min}^{\mathsf{divisor}} - 1$, and $N = q \cdot (q_{\min}^{\mathsf{divisor}} - 1) \cdot |w_{\mathsf{Fit}}|$.
3. *Create Equivalent EQC for $\lambda/\log(D)$ repetitions of the property testing protocol:*
   (a) Initialize an empty EQC instance $(\mathbb{F}_{p^*}, C_{\mathsf{PoSO}}, \mathrm{io}_{\mathsf{PoSO}})$ with $C_{\mathsf{PoSO}} = \emptyset$, $\mathrm{io}_{\mathsf{PoSO}} = \perp$, and existential wire values $w_{\mathsf{PoSO}} = w_{\mathsf{Fit}}$. More details in remark 2.
   (b) Sample vectors $\mathbf{r}^1, \ldots, \mathbf{r}^{\lambda/\log(D)}$, where each $\mathbf{r}^j \leftarrow [0, D)^{|w_{\mathsf{Fit}}|}$. Set $\mathrm{io}_{\mathsf{PoSO}} = \mathbf{r}^1 \| \ldots \| \mathbf{r}^{\lambda/\log(D)}$.
   (c) *Add Inner Product Constraints:*
   For each repetition $j \in [\lambda/\log(D)]$, append the following inner product constraint:
$$C_{\mathsf{PoSO}} \leftarrow C_{\mathsf{PoSO}} \cup \{v^j := < w_{\mathsf{PoSO}} \cdot \mathbf{r}^j >\}.$$
   (d) *Add Range Check Constraints:*
   Enforce that each $v^j < N$ by using bit-decomposition. For each v^j, introduce its bit-decomposition as existential wire values $b_1, \ldots, b_{\log(N)}$. Add the following constraints:
      i. For each bit $i \in [\log(N)]$, append the binary constraint:
$$C_{\mathsf{PoSO}} \leftarrow C_{\mathsf{PoSO}} \cup \{b_i^2 - b_i = 0\}.$$
      ii. Add the bit-decomposition constraint:
$$C_{\mathsf{PoSO}} \leftarrow C_{\mathsf{PoSO}} \cup \{v^j = \sum_{i=1}^{\log(N)} 2^i \cdot b_i\}.$$
   Although this is computationally expensive, notice that only one range check is needed per repetition.
4. Output the concatenated EQC instance $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}} \cup C_{\mathsf{PoSO}}, \mathrm{io}_{\mathsf{Fit}} \cup \mathrm{io}_{\mathsf{PoSO}})$.

Figure 5: Emulating Stage of FRE

the circuit arithmetizing this protocol, it must also be satisfiable. Thus, the augmented wires w_{Fit} make both $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}})$ and $(\mathbb{F}_{p^*}, C_{\mathsf{PoSO}}, \mathrm{io}_{\mathsf{PoSO}})$ satisfiable. □

Theorem 5 (Emulation Soundness)**.** *Assuming that $(q_{\min}^{\mathsf{divisor}})^{\mathsf{c}+2} \cdot q^2 \cdot |w_{\mathsf{Fit}}|^2 < p^*$, if the concatenated*

EQC instance $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}} \cup C_{\mathsf{PoSO}}, io_{\mathsf{Fit}} \cup io_{\mathsf{PoSO}})$ *is satisfiable, then one can extract a set of existential wires* $\tilde{w}_{\mathsf{Fit}}$ *that satisfy the original instance* $(\mathbb{Z}_q, C, io)$*, with an error probability of at most* $1/2^{\lambda}$*.*

Proof. Since the concatenated EQC is satisfiable and due to remark 2, there must exist existential wires w_{Fit} that make both $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}})$ and $(\mathbb{F}_{p^*}, C_{\mathsf{PoSO}}, \mathrm{io}_{\mathsf{PoSO}})$ satisfiable. The fact that $(\mathbb{F}_{p^*}, C_{\mathsf{PoSO}}, \mathrm{io}_{\mathsf{PoSO}})$ is satisfiable implies that the vector $w_{\mathsf{Fit}} \in \mathbb{F}_{p^*}^{|w_{\mathsf{Fit}}|}$ passes $\lambda/\log(D)$ repetitions of the property testing protocol with independently sampled random vectors $\mathbf{r}$. By the soundness of the property testing framework (claim 3), with an error probability of at most $1/2^{\lambda}$, the vector w_{Fit} belongs to the relation $\mathcal{R}_{N,D}$, where $D = q_{\min}^{\mathsf{divisor}} - 1$ and $N = q \cdot (q_{\min}^{\mathsf{divisor}} - 1) \cdot |w_{\mathsf{Fit}}|$.

Since w_{Fit} also makes $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}})$ satisfiable, this means that for each constraint $j \in [1, |C_{\mathsf{Fit}}|]$, the degree-2 constraint

$$\left(\sum_{a_i^j \in S^j} a_i^j\right) \cdot b^j = c^j + k^j \cdot q \mod p^*$$

is satisfiable, where $\{a_i^j\}, b^j, c^j, k^j \in \mathcal{R}_{N,D}$. Substituting each wire value with its rational representation yields:

$$\left(\sum_{a_i^j \in S^j} \frac{a_{i,1}^j}{a_{i,2}^j}\right) \cdot \frac{b_1^j}{b_2^j} = \frac{c_1^j}{c_2^j} + \frac{k_1^j}{k_2^j} \cdot q \mod p^*.$$

Let L denote the least common multiple (LCM) of denominators $\{a_{i,2}^j\}$ for $a_i^j \in S^j$. Multiplying both sides by $L \cdot b_2^j \cdot c_2^j \cdot k_2^j$, we get:

$$\left(\sum_{a_i^j \in S^j} \frac{L}{a_{i,2}^j} \cdot a_{i,1}^j\right) \cdot b_1^j \cdot c_2^j \cdot k_2^j = L \cdot b_2^j \cdot k_2^j \cdot c_1^j + L \cdot b_2^j \cdot c_2^j \cdot k_1^j \cdot q \mod p^*.$$

The LHS is an integer of at most $D^{|S|+1} \cdot N^2 \leq (q_{\min}^{\mathsf{divisor}})^{\mathsf{c}+2} \cdot q^2 \cdot |w_{\mathsf{Fit}}|^2$. The RHS is an integer of at most $D^{|S|+2} \cdot N \cdot q \leq (q_{\min}^{\mathsf{divisor}})^{\mathsf{c}+3} \cdot q^2 \cdot |w_{\mathsf{Fit}}|$. We assume that $q_{\min}^{\mathsf{divisor}} < |w_{\mathsf{Fit}}|$, so the LHS is always larger. Since $(q_{\min}^{\mathsf{divisor}})^{\mathsf{c}+2} \cdot q^2 \cdot |w_{\mathsf{Fit}}|^2 < p^*$, both LHS and RHS are less than p^* as integers. Thus, the equation holds over the integers, not just over $\mathbb{F}_{p^*}$. Dividing both sides by $L \cdot b_2^j \cdot c_2^j \cdot k_2^j$, it is easy to see that the equation holds over the rationals.

Finally, since $D < q_{\min}^{\mathsf{divisor}}$, by the q-invertible criterion (lemma 1), every rational in the equation is q-invertible. Therefore, we can cast each rational into a representation of $\mathbb{Z}_q$ elements. Let $\{\tilde{a}_i^j\}, \tilde{b}^j, \tilde{c}^j, \tilde{k}^j \in \mathbb{Z}_q$ be the casted elements. Now observe that:

$$\left(\sum_{\tilde{a}_i^j \in S^j} \tilde{a}_i^j\right) \cdot \tilde{b}^j = \tilde{c}^j \mod q.$$

Hence, we have extracted a set of wire values $\{\tilde{a}_i^j\}, \tilde{b}^j, \tilde{c}^j, \tilde{k}^j \in \mathbb{Z}_q$ that satisfy the jth constraint of the original EQC instance $(\mathbb{Z}_q, C, \mathrm{io})$.

In general, we denote by $\tilde{w}_{\mathsf{Fit}} \in \mathbb{Z}_q^{|w_{\mathsf{Fit}}|}$ the full set of casted existential wire values. By similar reasoning, we conclude that these wire values satisfy all the wiring constraints in the original EQC. ☐

6 FREPack: More Prover-Efficient SNARK Frontend for Highly Repetitive Computations

In this section, we propose **F**ast **R**ing **E**mulation-based **Pack**ing (FREPack), a combined SNARK frontend compiler designed to construct more prover-efficient commit-and-prove SNARKs for highly repetitive computations.

We outline the essential steps and components necessary before applying the FREPack frontend:

Preparing Commit-and-Prove SNARK Backends: Let $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ be any commit-and-prove SNARK backend for arbitrary relations (see definition 2) associated with a vector commitment scheme over the native field $\mathbb{F}_{p^*}$.

Reduction to SIMD EQC: As discussed in section 3.1, any highly repetitive computation can be reduced to a SIMD EQC with minimal additional wiring constraints, using standard SNARK frontends. For simplicity, we assume this reduction yields some SIMD EQC relation $\mathcal{R}^{C,\mathsf{SIMD}}$ with ℓ copies. We assume WLOG that the EQC is degree-2, defined over the native field $\mathbb{F}_{p^*}$, and $p_{\min}$-satisfiable with constant fan-in $\mathbf{c}$.

6.1 How to Apply FREPack SNARK Frontend

In Figure 6, we demonstrate how to apply the FREPack frontend to any commit-and-prove SNARK backend, producing prover-efficient SNARKs for any SIMD relation $\mathcal{R}^{C,\mathsf{SIMD}}$. Since both Pack and FRE allow independent compilation of the EQC constraints C and the input/output values io, when only compiling constraints, we set $\mathrm{io} = \perp$ to indicate the absence of input/output values.

<u>Prover Efficient Commit-and-Prove SNARK for Relation $\mathcal{R}^{C,\mathsf{SIMD}}$</u>

- $\mathsf{Gen}(1^\lambda, \mathcal{R}^{C,\mathsf{SIMD}}) \to (\mathsf{pk}, \mathsf{vk})$:
  1. <u>Pack:</u> Compress the SIMD relation $\mathcal{R}^{C,\mathsf{SIMD}}$: $(\mathbb{Z}_q, C, \perp) \leftarrow \mathsf{Pack}(\{(\mathbb{F}_{p^*}, C, \perp)\}_{i\in[\ell]})$.
  2. <u>FRE *Fitting:*</u> Let $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \perp) \leftarrow \mathsf{FRE.Fit}(\mathbb{Z}_q, C, \perp)$ be the fitted instance. Assert that $q_{\min}^{\mathsf{divisor}} = p_{\min}$ and $(p_{\min})^{\mathbf{c}+2} \cdot q^2 \cdot |w_{\mathsf{Fit}}|^2 < p^*$.
  3. <u>FRE *Emulating:*</u> Let $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}} \cup C_{\mathsf{PoSO}}, \perp) \leftarrow \mathsf{FRE.Emulate}(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \perp)$.
  4. <u>*Generating Proving Keys:*</u> Define two commit-and-prove relations $\mathcal{R}_{\mathsf{ck}}^{C_{\mathsf{Fit}}}$ and $\mathcal{R}_{\mathsf{ck}}^{\mathsf{PoSO}}$. Generate the proving/verifying key as $(\widetilde{\mathsf{pk}}, \widetilde{\mathsf{vk}}) \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{R}_{\mathsf{ck}}^{C_{\mathsf{Fit}}})$, $(\overline{\mathsf{pk}}, \overline{\mathsf{vk}}) \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{R}_{\mathsf{ck}}^{C_{\mathsf{PoSO}}})$. Then set $\mathsf{pk} = (\widetilde{\mathsf{pk}}, \overline{\mathsf{pk}})$ and $\mathsf{vk} = (\widetilde{\mathsf{vk}}, \overline{\mathsf{vk}})$.
- $\mathsf{Commit}(\mathsf{ck}, \{w_i\}_{i\in[\ell]}) \to c$:
  1. Pack existential wires into $w \leftarrow \mathsf{Pack}(\{w_i\}_{i\in[\ell]})$, and then update $w \to w_{\mathsf{Fit}}$.
  2. Commit to updated wires $c \leftarrow \mathsf{Commit}(\mathsf{ck}, w_{\mathsf{Fit}})$.
- $\mathsf{Prove}(\mathsf{pk}, \{\mathrm{io}_i\}_{i\in[\ell]}, \{w_i\}_{i\in[\ell]}) \to \pi$:
  1. Derive w_{Fit} from $\{w_i\}_{i\in[\ell]}$ as described in Commit. Similarly, derive $\mathrm{io}_{\mathsf{Fit}}$ from $\{\mathrm{io}_i\}_{i\in[\ell]}$. Then parse $\mathsf{pk} = (\widetilde{\mathsf{pk}}, \overline{\mathsf{pk}})$.
  2. <u>*Deriving randomness $\mathrm{io}_{\mathsf{PoSO}}$ via Fiat-Shamir:*</u> Set $\mathrm{io}_{\mathsf{PoSO}} \leftarrow \mathcal{H}(c)$, where $\mathcal{H}$ is a cryptographic hash function modeled as a random oracle.
  3. <u>*Proof Generation:*</u> W.r.t. commitment c, generate $\widetilde{\pi} \leftarrow \mathsf{Prove}(\widetilde{\mathsf{pk}}, (\mathrm{io}_{\mathsf{Fit}}, c), w_{\mathsf{Fit}})$ and $\overline{\pi} \leftarrow \mathsf{Prove}(\overline{\mathsf{pk}}, (\mathrm{io}_{\mathsf{PoSO}}, c), w_{\mathsf{Fit}})$, and set $\pi = (c, \widetilde{\pi}, \overline{\pi})$.
- $\mathsf{Verify}(\mathsf{vk}, \{\mathrm{io}_i\}_{i\in[\ell]}, \pi) \to \{0, 1\}$:
  1. Derive $\mathrm{io}_{\mathsf{Fit}}$ from $\{\mathrm{io}_i\}_{i\in[\ell]}$ as described in Prove, and then parse $\mathsf{vk} = (\widetilde{\mathsf{vk}}, \overline{\mathsf{vk}})$ and $\pi = (c, \widetilde{\pi}, \overline{\pi})$.
  2. <u>*Deriving Randomness $\mathrm{io}_{\mathsf{PoSO}}$ via Fiat-Shamir:*</u> Set $\mathrm{io}_{\mathsf{PoSO}} \leftarrow \mathcal{H}(c)$.
  3. <u>*Final Verification:*</u> Accept if and only if $\mathsf{Verify}(\widetilde{\mathsf{vk}}, (\mathrm{io}_{\mathsf{Fit}}, c), \widetilde{\pi}) = 1$ and $\mathsf{Verify}(\overline{\mathsf{vk}}, (\mathrm{io}_{\mathsf{PoSO}}, c), \overline{\pi}) = 1$.

Figure 6: Commit-and-Prove SNARK for Highly Repetitive Computations

Completeness: Assuming that the instances and existential wires $\left(\{\mathrm{io}_i\}_{i\in[N]}, \{w_i\}_{i\in N}\right)$ are in relation $\mathcal{R}^{C,\mathsf{SIMD}}$, we argue that the prover can generate accepting proof π with probability 1.

Since for all $i \in [N]$, the existential wires w_i satisfy the ith EQC instance $(\mathbb{F}_{p^*}, C, \mathrm{io}_i)$, by completeness of compiler Pack (claim 1), the packed wires w must satisfy the packed EQC instance $(\mathbb{Z}_q, C, \mathrm{io})$. Moreover, due to the completeness of FRE (theorem 4), the augmented wires w_{Fit} make both $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}})$ and $(\mathbb{F}_{p^*}, C_{\mathsf{PoSO}}, \mathrm{io}_{\mathsf{PoSO}})$ satisfiable. In other words, we have $((\mathrm{io}_{\mathsf{Fit}}, c), w_{\mathsf{Fit}}) \in \mathcal{R}_{\mathsf{ck}}^{C_{\mathsf{Fit}}}$ and $((\mathrm{io}_{\mathsf{PoSO}}, c), w_{\mathsf{Fit}}) \in \mathcal{R}_{\mathsf{ck}}^{C_{\mathsf{PoSO}}}$. Finally, due to the completeness of underlying commit-and-prove SNARK, the prover can generate accepting proofs $(\overline{\pi}, \widetilde{\pi})$ for both relations $\mathcal{R}_{\mathsf{ck}}^{C_{\mathsf{Fit}}}$ and $\mathcal{R}_{\mathsf{ck}}^{C_{\mathsf{PoSO}}}$ with probability 1.

Knowledge Soundness: We show that for any prover who generates accepting proofs π, one can extract a set of valid witness $\{w_i\}_{i\in N}$ such that $\left(\{\mathrm{io}_i\}_{i\in[N]}, \{w_i\}_{i\in N}\right)$ are in relation $\mathcal{R}^{C,\mathsf{SIMD}}$ with all but probability $\mathsf{negl}(\lambda)$.

Due to the knowledge soundness of underlying commit-and-prove SNARK, with all but probability $\mathsf{negl}(\lambda)$, we can extract valid witness w_{Fit} which make both $(\mathbb{F}_{p^*}, C_{\mathsf{Fit}}, \mathrm{io}_{\mathsf{Fit}})$ and $(\mathbb{F}_{p^*}, C_{\mathsf{PoSO}}, \mathrm{io}_{\mathsf{PoSO}})$ satisfiable. Furthermore, by construction of Pack, we have $q_{\min}^{\mathsf{divisor}} = p_{\min}$. Since $(p_{\min})^{\mathsf{c}+2} \cdot q^2 \cdot |w_{\mathsf{Emulate}}|^2 < p^*$, by soundness of FRE (theorem 5), with all but probability $1/2^{\lambda} = \mathsf{negl}(\lambda)$, one can extract a set of existential wires $\tilde{w}_{\mathsf{Emulate}}$ which satisfy the instance $(\mathbb{Z}_q, C, \mathrm{io})$. Finally, since every EQC instance $(\mathbb{F}_{p^*}, C, \mathrm{io}_i)$ is $p_{\min}$-satisfiable, by soundness of compiler Pack (claim 1), we can further extract N set of existential wires $\{w_i\}_{i\in N}$ such that each w_i satisfies the ith EQC instance $(\mathbb{F}_{p^*}, C, \mathrm{io}_i)$. Therefore, we have $\left(\{\mathrm{io}_i\}_{i\in[N]}, \{w_i\}_{i\in N}\right)$ in relation $\mathcal{R}^{C,\mathsf{SIMD}}$.

Succinct Verification: We achieve succinct proofs due to the succinctness of the commitment scheme and the commit-and-prove SNARK. Nonetheless, the verifier's runtime remains linear in the length of the prover's witness. Specifically, in the FRE emulation stage, the verifier must hash and generate $\lambda/\log(D)$ random vectors as the input $\mathrm{io}_{\mathsf{PoSO}}$, with each vector having a length of $|w_{\mathsf{Fit}}|$. This becomes impractical in scenarios that require succinct verification, where the verifier's runtime should be bounded by $O(p(\lambda))$ for some fixed polynomial $p(\cdot)$, independent of witness size.

To address this, we reuse randomness during the emulation stage. Instead of executing the Batch-PoSO protocol over the entire vector w_{Fit}, we split it into $d = \frac{|w_{\mathsf{Fit}}|}{p(\lambda)}$ chunks, where each chunk has size $p(\lambda)$. Let $w_{\mathsf{Fit}} = w_{\mathsf{Fit}}^1 || \dots || w_{\mathsf{Fit}}^d$ represent the split witness vector. We then execute the Batch-PoSO protocol for each chunk w_{Fit}^i $(i \in [d])$. Crucially, the verifier samples only one random vector $\mathbf{r} \in \mathbb{Z}_D^{p(\lambda)}$ and reuses this vector across all d Batch-PoSO executions. By the union bound, this introduces a soundness error of at most d/D. To amplify the soundness to $1/2^{\lambda}$, we repeat the Batch-PoSO protocol $\frac{\lambda}{\log(D)-\log(d)}$ times. This reduces the randomness complexity to $\frac{\lambda}{\log(D)-\log(d)} \cdot p(\lambda) \approx O(p(\lambda))$, thus achieving succinct verification.

Honest Verifier Zero-Knowledge: Since both frontend compilers do not require the prover's witness as public input, if the underlying commit-and-prove SNARK satisfies honest verifier zero-knowledge, the resulting SNARK will also preserve this property.

7 Implementations and Evaluations

We implement $\mathsf{FREPack}$ and evaluate its performance by instantiating it with concrete SNARK frontend compilers and commit-and-prove backends. For this evaluation, we focus on our motivating scenario—generating *constant-size proofs* for a large number of hash evaluations, as found in real-world applications such as *zkRollup* and proof recursion. We mainly compare our frontend compiler with lookup arguments, the only alternative for frontend optimization in highly repetitive computations.

7.1 Setup and Parameters

Base Frontend and Backends: We use circom, a widely adopted SNARK frontend compiler, as the base (naive) frontend. This frontend outputs circuit representations in the form of Rank-1 Constraint Systems (R1CS). Our $\mathsf{FREPack}$ builds on top of `circom` to produce more efficient R1CS circuits for repetitive computations.

Given the requirement of constant-size proofs, we only focus on all group-based backends, as so far they are the only viable option for ensuring constant proof sizes. We evaluate against popular group-based commit-and-prove backends, including Spartan [Set20] and a commit-and-prove variant of the well-known Groth16 [Gro16], as described in [CFQ19]. Additionally, to demonstrate that our frontend improvements *complement* rather than overlap with existing backend optimizations on any algorithmic level, we include Nova [KST22], an algorithmically optimized, group-based backend specifically designed for incremental computations[4].

Circuits and Packing Factor: To align with our motivating applications, we evaluate the performance of SHA2-256 and SHA3-512, the primiary hash functions used in zkRollups and proof recursions[5]. Additionally, to demonstrate the generality of our method, we include random binary/arithmetic programs. These random programs lack the small repetitive structures characteristic of SHA programs, making them unsuitable for frontend optimizations like lookup arguments.

We use the elliptic curve group `BLS12-381`, which has an order of $p^* \approx 2^{255}$. According to theorem 5, for any packing factor ℓ and $q = \prod_{i=1}^{\ell} q_i$, we require $(q_{\min}^{\mathsf{divisor}})^{\mathsf{c}+2} \cdot q^2 \cdot |w_{\mathsf{Fit}}|^2 < p^*$. Thus in order to determine the optimal packing factor ℓ, we first bound the size of q and express it as the product of as many distinct primes as possible, while ensuring all primes are larger than $q_{\min}^{\mathsf{divisor}}$.

For both SHA2 and SHA3 circuits (with slight modifications), we observe that $q_{\min}^{\mathsf{divisor}} = p_{\min} < 2^8$, $\mathbf{c} = 3$, and $|w_{\mathsf{Fit}}| < 2^{16}$. This results in $q < 2^{97}$, allowing q to be represented as a product of up to 12 distinct primes, all larger than $q_{\min}^{\mathsf{divisor}}$. Thus, we set our packing factor ℓ to 12. For a soundness error less than 2^{-128}, we repeat its property testing framework 17 times.

Measurements and Hardware Setup: In our experiments, we evaluate the efficiency of each frontend by measuring three key metrics of the circuits they generate: the number of constraints ($|C|$), the number of witness variables ($|w|$), and the backend prover runtime required for each backend. These experiments were run on a machine with Intel Xeon Skylake 6130 CPU (2.1 GHz) and 96GB of RAM.

7.2 Experiment Design

Baselines: We compare FREPack against two baseline frontend compilers. The first baseline is the 'naive' approach, where we use circom directly out-of-the-box. The second baseline is the lookup argument, powered by the LogUp method [Hab22]. Since no existing R1CS circuit implements a lookup-based version of SHA2/SHA3, we designed our own version of SHA2, albeit slightly unfaithful. We use tables of size 2^8 and break each word into 8 nibbles of size 2^4. While all other operations in SHA2 are implemented faithfully, we simplify the Σ (rotation) operation by simulating it with 8 random tables, as it is complex to represent with actual lookup tables. This simplification in fact reduces the overall circuit size, giving an advantage to the lookup argument approach when it comes to benchmarking.

We implement lookup arguments within circom so as to generate R1CS circuits. Notably, LogUp method represents the most efficient, *backend-universal* implementation of lookup arguments. While further optimizations of LogUp [PH23] and other efficient lookup arguments such as [STW24] exist, these methods require specialized backends and do not meet constant proof sizes without resorting to recursion, which introduces prohibitively expensive overheads and would be significantly slower.

Experiments: To simulate Merkle tree opening proofs in zkRollup and similar applications, our first experiment focuses on incremental computations, specifically iterative SHA2 or SHA3 hashing. More precisely, we consider $n \cdot \ell$ iterations of the SHA program, where ℓ is our packing factor and each iteration corresponds to hashing a single block of a message. The value of n ranges from 1 to 24. For each configuration, we compare the R1CS circuits generated by FREPack against the two baseline compilers, benchmarking the aforementioned cost metrics.

Our second experiment evaluates the SIMD computation of random binary/arithmetic circuits. In this case, we begin by generating random circuits of varying sizes and then evaluate ℓ copies of each circuit over different inputs. We only benchmark the prover time in this experiment.

[4]Integrating Nova backend with FREPack requires minor tweaks. Intuitively, we first apply FREPack and then perform the folding, hence effectively reducing the number of foldings required by a factor fo ℓ. We defer the details to the full version.

[5]Although some zk-friendly hashing such as Poseidon is sometimes used as replacement, it is a relatively non-standard construction and only provides heuristic security guarantees. So we do not consider them in this work.

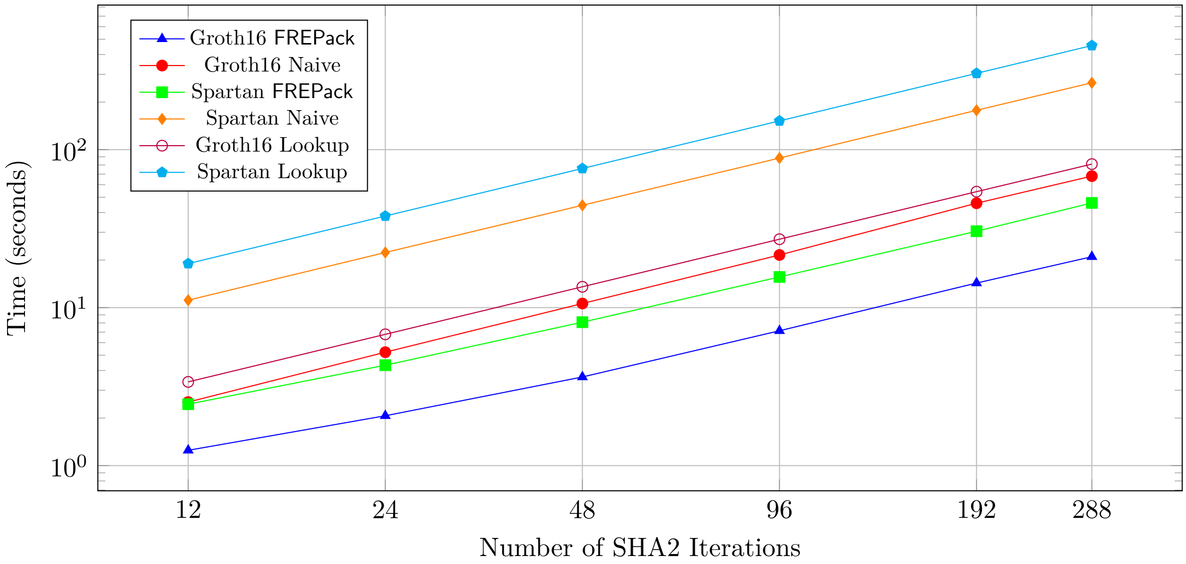

Figure 7: Backend Prover Runtime for SHA-2 Experiment

7.3 Evaluations

7.3.1 Experiment I: Merkle Opening

We present the circuit size measurements for iterative SHA2-256 hashing in a Merkle opening, as summarized in section 7.3.2. For a large number of iterations, compared to the naive frontend, the lookup method reduces constraints by 1.9× but increases variables by 1.7×. In contrast, FREPack reduces both constraints by 8.5× and variables by 2.6×. These reductions directly lower backend prover costs: FREPack improves Groth16 prover time by 3.6×, Nova by 3.8× and Spartan by 5.9×, as shown in fig. 7 and fig. 10 in appendix. The lookup method, however, underperforms even compared to the naive approach due to prover committing to more variables.

7.3.2 Experiment II: Random Circuits

We report the prover runtimes for random arithmetic circuits in fig. 8 and random binary circuits in fig. 9 in appendix, with varying circuit sizes based on the number of constraints. As one can see, FREPack reduces the Groth16 prover cost by 4× and Spartan by 6.4×.

Iterations	**12**	**24**	**48**	**96**	**192**	**288**
Constraints						
Naive	371424	742848	1485696	2971392	5942784	8914176
Lookup	193152	386304	772608	1545216	3090432	4635648
FREPack	45470	88866	175658	351316	702632	1051874
Variables						
Naive	371425	742849	1485697	2971393	5942785	8914177
Lookup	622860	1245720	2491440	4982880	9965760	14948640
FREPack	308210	444360	716660	1263317	2356629	3447888

Table 1: Comparison of Constraints and Variables for SHA2

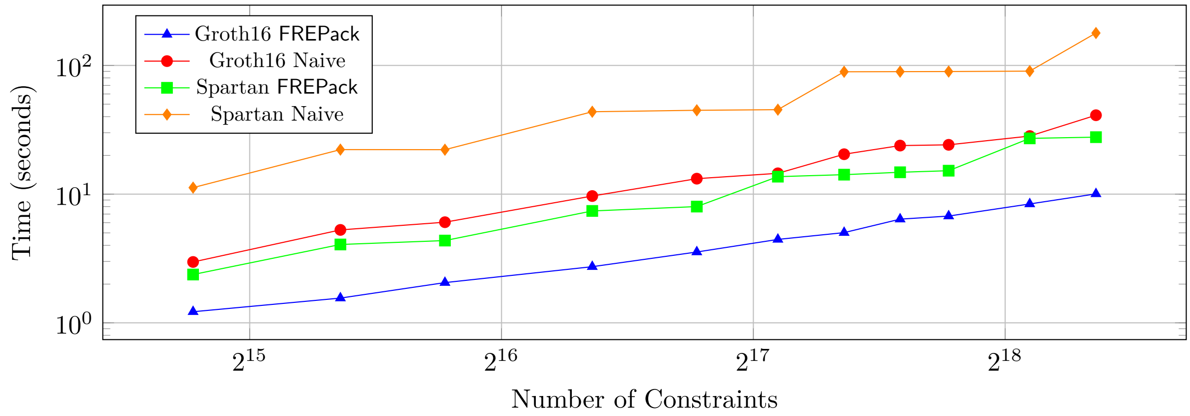

Figure 8: Backend Prover Runtime for Random Arithmetic Circuits

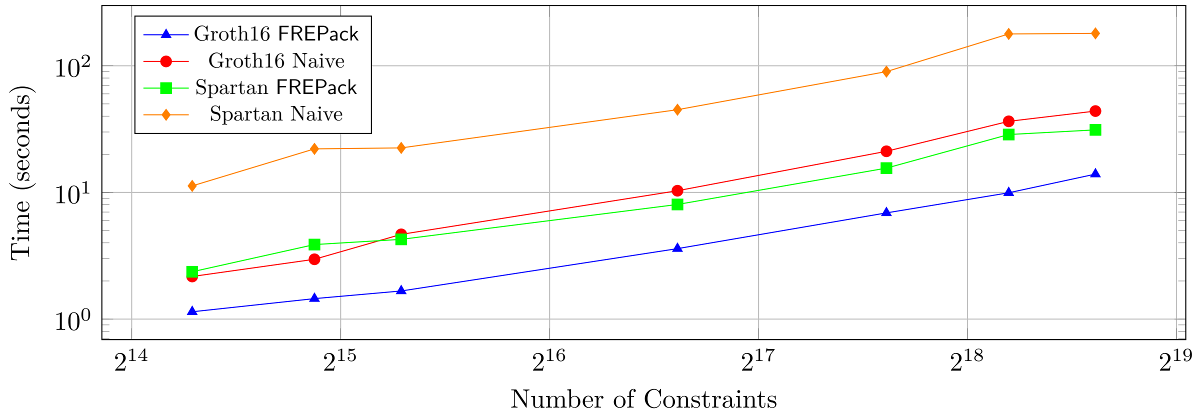

Figure 9: Backend Prover Runtime for Random Binary Circuits

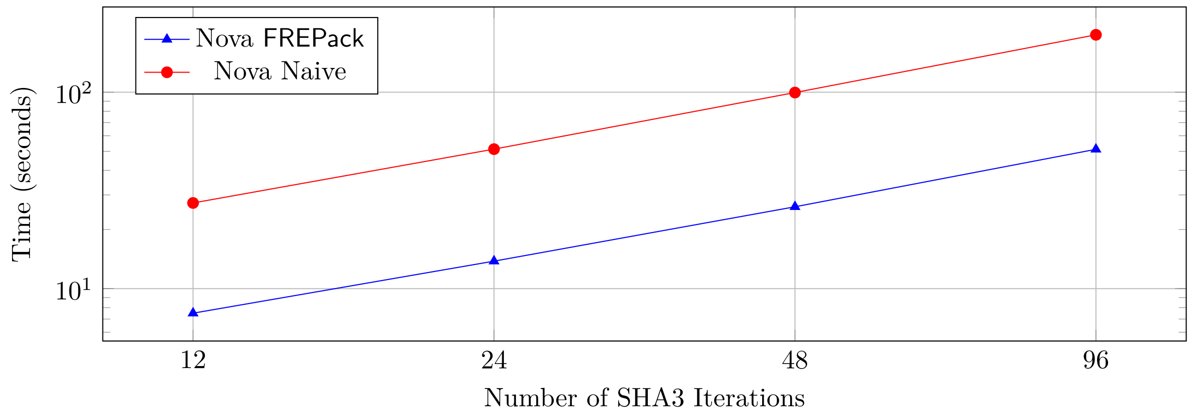

Figure 10: Backend Prover Runtime for SHA-3 Experiment

References

[AGL+23] Arasu Arun, Chaya Ganesh, Satya Lokam, Tushar Mopuri, and Sriram Sridhar. Dew: A transparent constant-sized polynomial commitment scheme. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *Public-Key Cryptography – PKC 2023*, pages 542–571, Cham, 2023. Springer Nature Switzerland.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

[BC23] Benedikt Bünz and Binyi Chen. Protostar: Generic efficient accumulation/folding for special sound protocols. Cryptology ePrint Archive, Paper 2023/620, 2023. `https://eprint.iacr.org/2023/620`.

[BCGL22] Jonathan Bootle, Alessandro Chiesa, Ziyi Guan, and Siqi Liu. Linear-time probabilistic proofs with sublinear verification for algebraic automata over every field. Cryptology ePrint Archive, Paper 2022/1056, 2022. `https://eprint.iacr.org/2022/1056`.

[BCR+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Heidelberg, May 2019.

[BFS20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 677–706. Springer, Heidelberg, May 2020.

[CBBZ22] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. Cryptology ePrint Archive, Paper 2022/1355, 2022. `https://eprint.iacr.org/2022/1355`.

[CFKS22] Hien Chu, Dario Fiore, Dimitris Kolonelos, and Dominique Schröder. Inner product functional commitments with constant-size public parameters and openings. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks*, pages 639–662, Cham, 2022. Springer International Publishing.

[CFQ19] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2075–2092. ACM Press, November 2019.

[CGKR22] Geoffroy Couteau, Dahmun Goudarzi, Michael Klooß, and Michael Reichle. Sharp: Short relaxed range proofs. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 609–622. ACM Press, November 2022.

[cir] circom. `https://github.com/iden3/circom`.

[CKLR21] Geoffroy Couteau, Michael Klooß, Huang Lin, and Michael Reichle. Efficient range proofs with transparent setup from bounded integer commitments. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part III*, volume 12698 of *LNCS*, pages 247–277. Springer, Heidelberg, October 2021.

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, *EURO-*

CRYPT 2020, Part I, volume 12105 of *LNCS*, pages 769–793. Springer, Heidelberg, May 2020.

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

[DP24] Benjamin E. Diamond and Jim Posen. Polylogarithmic proofs for multilinears over binary towers. Cryptology ePrint Archive, Paper 2024/504, 2024. https://eprint.iacr.org/2024/504.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

[GJJZ22] Sanjam Garg, Abhishek Jain, Zhengzhong Jin, and Yinuo Zhang. Succinct zero knowledge for floating point computations. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1203–1216. ACM Press, November 2022.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 39–56. Springer, Heidelberg, August 2008.

[GLS+21] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and post-quantum snarks for r1cs. Cryptology ePrint Archive, Paper 2021/1043, 2021. https://eprint.iacr.org/2021/1043.

[GNSV21] Chaya Ganesh, Anca Nitulescu, and Eduardo Soria-Vazquez. Rinocchio: Snarks for ring arithmetic. Cryptology ePrint Archive, Paper 2021/322, 2021. https://eprint.iacr.org/2021/322.

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021. https://eprint.iacr.org/2021/1063.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

[GW20] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Paper 2020/315, 2020. https://eprint.iacr.org/2020/315.

[Hab22] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530, 2022. https://eprint.iacr.org/2022/1530.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

[KPS18] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 944–961, 2018.

[KS23] Abhiram Kothapalli and Srinath Setty. Hypernova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive, Paper 2023/573, 2023. https://eprint.iacr.org/2023/573.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 359–388. Springer, Heidelberg, August 2022.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010.

[OBW22] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2248–2266, 2022.

[PH23] Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using GKR. Cryptology ePrint Archive, Paper 2023/1284, 2023.

[rol] circom.

[RZR22] Noga Ron-Zewi and Ron D. Rothblum. Proving as fast as computing: Succinct arguments with constant prover overhead. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, page 1353–1363, New York, NY, USA, 2022. Association for Computing Machinery.

[SB23] István András Seres and Péter Burcsi. Behemoth: transparent polynomial commitment scheme with constant opening proof size and verifier time. Cryptology ePrint Archive, Paper 2023/670, 2023. https://eprint.iacr.org/2023/670.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Heidelberg, August 2020.

[Sta21] StarkWare. ethSTARK documentation. Cryptology ePrint Archive, Paper 2021/582, 2021.

[STW24] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with lasso. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 180–209, Cham, 2024. Springer Nature Switzerland.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 71–89. Springer, Heidelberg, August 2013.

[Tha23] Justin Thaler. Proofs, arguments, and zero-knowledge, 2023.

[WTS+18] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-efficient zksnarks without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 926–943, Los Alamitos, CA, USA, may 2018. IEEE Computer Society.

[XL24] Tiancheng Xie and Tianyi Liu. Almost optimal succinct arguments for boolean circuit on RAM. Cryptology ePrint Archive, Paper 2024/839, 2024. https://eprint.iacr.org/2024/839.

[XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 299–328. Springer, Heidelberg, August 2022.

[ZBK+22] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 3121–3134. ACM Press, November 2022.

[ZLW+21] Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinuo Zhang, Dawn Song, Xiang Xie, and Yupeng Zhang. Doubly efficient interactive proofs for general arithmetic circuits with linear prover time. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 159–177. ACM Press, November 2021.

[ZXZS20] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy*, pages 859–876. IEEE Computer Society Press, May 2020.

8 Appendix

8.1 Recursion Costs of Hash-based Backends

Since hash-based commit-and-prove SNARK backends do not directly produce constant-size proofs, a group-based backend is often employed to recursively prove the verifier of these hash-based systems. This step, known as proof recursion, can be computationally expensive.

We observe that most existing hash-based backends use an IOP-based framework, where the verifier's dominant task is verifying Merkle tree openings. These arise from the verifier's queries to the prover's committed messages in any IOP. Among hash-based backends, the most efficient one requires the verifier to make 55 queries for 128-bit security (assuming all Reed-Solomon code conjectures) and 140 queries without such conjectures [Sta21]. Each query typically involves verifying a Merkle opening path of length $\log(n) \cdot \log(1/\rho) + \log^2(n)$, where ρ is the code rate used in the IOP, and n is roughly the size of the circuit to be proved.

Now suppose our goal is to generate a constant-size proof for n hash evaluations using proof recursion. Plugging in minimum parameters used in industry, the recursion cost is at least evaluating $55 \cdot (\log(1000 \cdot n) \cdot 4 + \log^2(1000 \cdot n))$ hashes, which is only smaller than n when $n > 2^{15}$. For provable security (without any conjectures), the cutoff threshold even exceeds 2^{16}. Therefore, unless one aims to prove more than these number of hashes, the hashes need to be proved in recursion alone exceeds the number of hashes one starts with. The implication is that in all practical scenarios such as zkRollup, it is much faster to prove these hashes directly using a group-based backend rather than first using a hash-based backend and then recursively proving with a group-based backend.