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 usually feature 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 the circuit may be defined over small fields, the backend prover often needs to lift the computation to much larger fields for achieving soundness. This gap results in concrete overheads, for example, when proving SHA2 programs with pairing-based SNARKs.

For a class of computations that are *highly repetitive*, we propose an improved frontend that partially bridges this gap. Compared with existing works, our frontend yields circuit representations defined over larger fields but of smaller size. Our implementation shows that for ≈ 100 iterations of SHA2-256 instances, our improved frontend boosts prover runtime by over $3.8\times$.

Central to our result and of independent interest, is an efficient technique for proving non-native ring arithmetic.

1 Introduction

Succinct arguments [Kil92] allow a prover to convince a verifier that an $\mathcal{NP}$ statement is true through some interactive protocols, with communication and verifier running time sub-linear in the size of the prover's witness. Soundness requires that no computationally bounded prover can convince a verifier of a false statement. Those arguments can also be paired with an additional *zero knowledge* (ZK) property [GMR85], which requires that the verifier does not learn anything beyond the veracity of the statement. Zero-knowledge Succinct Non-interactive ARguments of Knowledge (zkSNARKs) are succinct arguments which are zero-knowledge and do no involve any interactions. Enjoying all these great properties, zkSNARKs have found numerous real-world applications, e.g., in the design of blockchains [BCG+14]. This has led to extensive research towards improving the efficiency of zkSNARKs in practice [CMT12, Tha13, AHIV17, BBB+18, XZZ+19, BCR+19, Set20, COS20, BFS20, ZXZS20, ZLW+21], both in terms of prover and verifier running times. As the time of writing, the best SNARK prover running time is already asymptotically linear in the size of the statement [GLS+21, XZS22, CBBZ22] while still maintaining sublinear verifier running time.

Modern SNARK Paradigm: Frontend v.s. Backend Modern SNARKs are general-purpose and engineered to prove the correctness of any computer programs via following two steps:

First, the program is complied into a special kind of circuit representation which is amenable to the current SNARK technology. This process is often called the SNARK **frontend**. Importantly, the satisfiability of such circuit should reflect the correct evaluation of the computer program. In practice, this circuit representation resembles the original program and they often share the same input and output values.

Subsequently, this circuit representation is fed into a SNARK which is specifically designed to prove *circuit satisfiability*. This process is often called the SNARK **backend**. Most aforementioned constructions of SNARKs are examples of such. For example, some popular ones [Gro16, CHM+20, BCR+19] target a special circuit representation called Rank-1 Constraint Systems (R1CS).

Measuring Prover Efficiency The prover efficiency remains a core bottleneck in the real-world deployment of SNARKs. In the view of above paradigm, we can separate our consideration with respect to the frontend and backend.

Frontend Efficiency is mainly measured by the size of the circuit representation, relative to the size of the original computer program. For example, in R1CS, this size is determined by three main characteristics: the length and width (hence dimension) of the matrices, and the number of non-zero entries in these matrices.

Backend Efficiency is almost always dictated by the number of *Cryptographic Operations* required. For example, in [Gro16], those operations correspond to multi-scalar exponentiations (MSM) in an elliptic curve group of large order. One can abstractly view those cryptographic operations as computing arithmetics over some very large field. Furthermore, this number is usually proportional to the frontend size.

A Dichotomy of Fields Interestingly, there exists an inconspicuous dichotomy between frontend and backend: For most computer programs, the circuit representations outputted by the frontend are naturally defined over some small finite fields. Its size is comparable to the size of the variables living inside the computer program. For example, the circuit representation of the SHA2-256 binary computation can be defined over a field of size $\approx 2^8$ (for example, $\mathbb{F}_{253}$). Nonetheless, for the sake of soundness, the backend field must be very large. For example, [Gro16] uses the field $\mathbb{F}_{p^*}$ where p^* is around 255 bit prime, hence orders of magnitude larger than $\mathbb{F}_{253}$.

This size dichotomy is often solved by simply embedding the circuit representation defined over small fields inside the larger field. When the two fields substantially differ in size (as for most computer programs), this creates a **noticeable inefficiency.** For example, consider the circuit which checks that a bit b takes binary values: $b^2 - b = 0$. It is indeed sufficient to check for this arithmetic relation over any non-trivial field. Yet the backend prover needs to treat this relation over certain gigantic field $\mathbb{F}_{p^*}$ and subsequently perform multiplications by large random numbers in this field in order to argue this relation. Conceptually speaking, a significant portion of field $\mathbb{F}_{p^*}$ is wasted.

In this work we seek to incorporate this aspect into frontend design consideration. More specifically, for any circuit representation defined over some small field, we wonder if one can **'squeeze'** the circuit at the expense of **expanding** the field that it must be defined over. Since the prover must pay for the cost of large field either way, a smaller-sized circuit would strictly improve its performance. This motivates us to study the following question:

For certain programs, can we improve their frontend efficiency by exploiting full power of backend field?

1.1 Our Contributions

In this work we answer above question positively for a special class of programs which are *highly repetitive*. Informally speaking, a computation is highly repetitive if the same sub-computation is repeated multiple times with different inputs. One ubiquitous example is data-parallel (SIMD) computation. More concretely, our main contributions are as follows:

Techniques for Packing Highly Repetitive Circuits We introduce a packing technique for verifying highly repetitive computations. More specifically, for any computation consisting of ℓ copies of sub-computations. Our packing technique allows us to reduce the size of overall circuit representation by ℓ-fold compared to that of naive representation, at the cost of defining it over some large and non-native ring. [1]

Techniques for Efficiently Proving Non-native Arithmetic Crucial to our main results, and of independent interests, is an information theoretic emulation technique which allows the prover to efficiently emulate arithmetic behaviors of a non-native ring $\mathbb{Z}_q$ inside some fixed prime field $\mathbb{F}_{p^*}$.

[1] Although our plain packing technique yields circuit size reduction by l-fold, in practice the reduction is also limited by the backend field size.

Improved Frontend for Highly Repetitive Computations We combine our techniques into an improved frontend compiler for highly repetitive computations. It can be integrated with a class of popular backends called commit-and-prove SNARKs. We implement our frontend and instantiate it with two commit-and-prove backends: Marlin [CHM+20] and Nova [KST22], and benchmark our improvements.

1.2 Applications

Our improved frontend can be used to speed up a number of zero-knowledge proof applications which involve highly repetitive computations. For example, in zkRollup [rol]/zk-EVM, a prover needs to prove the opening of some Merkle Tree commitment, which involves proving the knowledge of a root-to-leaf path which corresponds to some sequential hash computations (such as SHA3/keccak). As another example, in many Proof-of-Stake blockchains, the proof involves verifying hundred of signatures. In Cosmos, each corresponds to an EdDSA signature, whose verification involves computing SHA2-512 hash functions.

1.3 Related Works

We give a survey of related works which seek to mitigate the dichotomy of field between SNARK frontend and backend. Some of these techniques are "backend oriented" while others are "frontend oriented".

Backend Oriented There are a number of works aiming to allow the backend prover to run over any finite field (even very small fields) to improve prover's efficiency. The line of works [RZR22, BCGL22] achieve this by building some specific Interactive Oracle Proof (IOP). However, they currently only remain theoretically interesting since the verifier's running time and proof size are linear in [RZR22], and sublinear but still very large in [BCGL22]. Furthermore, these constructions only work with very specific circuit representations which are not used in practice. Other works [AHIV17, KKW18] try to achieve this via "MPC in the head", but these protocols also yield considerably large proof size. Finally, [WYKW21, WYY+22] considers VOLE-based efficient zero-knowledge proofs but does not achieve sublinear verification.

Frontend Oriented Look-up arguments [GW20, ZBK+22] aim to reduce the task of checking multiple bit-wise operations in computer programs with look-up of a single value in a large truth table. For example, the bit-wise XOR operation between two 16-bit strings is replaced with a table consisting of all 2^{32} possible outputs, each entry in the table being a 16-bit integer value. Look-up arguments can help frontend design as follows: Suppose in the circuit representation, a sub-computation involving multiple of binary gates is repeated frequently. Then one can batch these gates into a single "look-up gate" with respect to some table, thus reducing each sub-computation effectively into a "look-up gate". On one hand, since the values in this table are large, this "look-up gate" must be defined over larger fields. On the other hand, the size of the circuit becomes much smaller due to reduced number of gates.

We compare our work with look up arguments as follows: First, known lookup arguments are limited by the table size due to the expensive cost of cryptographically committing to the lookup table. For example, in [GW20], to perform m lookups into a table of size N, the prover has to commit to $5 * \max(m, N)$ field elements. [ZBK+22] pushes the bulk of this work into a pre-processing phase, but still requires prover committing to N field elements in the table and additional pre-processing involving $O(N \log(N))$ exponentiations in a cryptographic group. It also uses a large structured reference string (SRS) as big as the lookup table. In practice, the table size is often less than 2^{16}, thus allowing to 'pack' at most 8 binary gates. For example, for SHA2-256 program which operates on 32-bit words, each word operation needs to be further broken down into $8 * 4$ bits chunks in order to apply lookup arguments. In comparison, our method does not need a table and has more tolerance for 'packing'. For example, we can 'pack' 12 of SHA2-256 circuits easily. Furthermore, no one needs to pay for the cost of committing to the table as well as storing the large CRS required for commitment. Secondly, look-up gates are specialized to the Plonk-style [GWC19] circuit representation. This raises two concerns: 1. The use of look-up gates requires rewriting the circuit topology, thus prone to errors in implementation. 2. Only some specialized SNARKs are designed to work with such circuit representation. In comparison, our method can be naturally extended to support all circuit representations as well as preserving the circuit topology. This makes our method easier to implement, and readily compatible with almost all backends.

Other Optimizations on Highly Repetitive Computations The work of [Tha13] improves the techniques of [GKR08] in the setting of data-parallel (a.k.a. SIMD) computations. The work of [KST22, BC23, KS23] consider folding schemes for incrementally verifiable computation that improves prover's running time. Furthermore, the work of [XZC+22] leverages multiple provers to speed up proving SIMD computations. We point out that the improvements behind all these works do not rely on mitigating the field discrepancy, hence orthogonal to our contributions. In fact we show in section 8.2 that our improved frontend can be used in conjunction with those works so as to achieve a double prover-speed-up.

2 Preliminaries

Notation: We use λ for the security parameter and let $\mathsf{negl}(\lambda)$ denote a negligible function: That is, for all polynomial $p(\lambda)$, it holds that $\mathsf{negl}(\lambda) < 1/p(\lambda)$ for large enough λ. We use $\mathbf{z}$ to denote a vector, $\mathbf{z}[i]$ to denote the i^{th} element in $\mathbf{z}$. We use the notation $< \mathbf{z_1} \cdot \mathbf{z_2} >$ to denote the inner product between two vectors $\mathbf{z_1}$ and $\mathbf{z}_2$. For an integer n, we shall use $[n]$ for the set $\{1, 2, \ldots, n\}$. Let $p\mathbb{Z}$ be the ideal generated by some number $p \in \mathbb{Z}$ and correspondingly let $\mathbb{F}_p$ denote the field $\mathbb{Z}/p\mathbb{Z}$, which corresponds to all the integers modulo p. We use $\mathsf{diag}^m(a)$ to denote the m-by-m diagonal matrix where the values along the diagonal is filled with a. We use the term PPT to stand for all *efficient* adversaries which runs in probabilistic polynomial time in the security parameter λ. We sometimes refer to these algorithms as *efficient* algorithms.

2.1 Chinese Remainder Theorem

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

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

where the isomorphism is given by the mapping $f : \mathbb{Z}_q \to \mathbb{F}_{q_1} \times \cdots \times \mathbb{F}_{q_n}$ as follows:

$$f(a) = (a \mod q_1, \ldots, a \mod q_n).$$

where the inverse mapping $f^{-1} : \mathbb{F}_{q_1} \times \cdots \times \mathbb{F}_{q_n} \to \mathbb{Z}_q$ is given as follows:

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

Each coefficient λ_i is an integer such that

$$\lambda_i \mod q_i = 1 \qquad \text{and} \qquad \forall j \neq i,\ \lambda_i \mod q_j = 0.$$

Those integers can be efficiently computed as follows. Let $Q = \prod_{j \neq i} q_j$ be the product of q_j's except for q_i. Then,

$$\lambda_i = Q \cdot Q^{-1},$$

where Q^{-1} is such that $Q \cdot Q^{-1} = 1 \mod q_i$.

We formalize the above discussion into a set of packing interface:

Definition 2.1 (CRT Packing Scheme)**.** *Let there be a set of prime numbers $(q_1, \ldots, q_n)$ and let $q = \prod_{i \in [n]} q_i$. A **CRT Packing** Scheme with respect to this set consists of the two algorithms $(\mathsf{CRT.Pack}, \mathsf{CRT.Unpack})$ with the following syntax:*

- $\mathsf{CRT.Pack}(a_1, \ldots, a_n) \to a$*: The packing algorithm takes as input $a_i \in \mathbb{F}_{q_i}$ from each field, and packs them into a number $a \in \mathbb{Z}_q$.*
- $\mathsf{CRT.Unpack}(a) \to (a_1, \ldots, a_n)$*: The unpacking algorithm takes as input some number $a \in \mathbb{Z}_q$ and recovers a set of n numbers $(a_1, \ldots, a_n)$ where $a_i \in \mathbb{F}_{q_i}$ for each field $i \in [n]$.*

As described above, the packing and unpacking algorithm naturally correspond to the mapping function f induced by ring isomorphism $\mathbb{Z}_q \cong \mathbb{F}_{q_1} \times \cdots \times \mathbb{F}_{q_n}$.

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 commitment key generation algorithm take as input the security parameter, outputs a commitment key ck and specifies an allowed message space $\mathbb{F}_{p^*}^n$.
- $\mathsf{Commit}(\mathsf{ck}, \mathbf{z}) \to c$: The commitment algorithm takes as input a commitment key ck, an vector $\mathbf{z} \in \mathbb{F}_{p^*}^n$, and outputs a commitment c.

We require the following properties to hold:

Succinct Commitment The size of commitment c is independent of the length of vector n.

Binding: Due to the size shrinking of commitments, there must exist vectors which collide to the same commitment. Nevertheless, we require that for all efficient algorithm A, finding such collision is intractable:

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

Additive Homomorphic Vector Commitment. We observe that most of existing vector commitment scheme such as [BBB+18, KZG10] have the following "additive homomorphism" structure, which allows to any two commitments $\mathsf{Commit}(\mathsf{ck}, \mathbf{z_1})$, $\mathsf{Commit}(\mathsf{ck}, \mathbf{z_2})$ to be 'added' so as to obtain a new commitment $\mathsf{Commit}(\mathsf{ck}, \mathbf{z_1} + \mathbf{z_2})$.

2.3 Non-interactive Argument of Knowledge

We denote any relation by $\mathcal{R}(,\cdot,)$ and say that a pair of instance $\mathbb{X}$ and 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, the description of relation R, and outputs a (public) proving key pk as well as a verification key vk.
- $\mathsf{Prove}(\mathsf{pk}, \mathbb{X}, w) \to \pi$: The proving algorithm takes as input the proving key pk, the instance $\mathbb{X}$ and some alleged witness w, and outputs some proof π.
- $\mathsf{Verify}(\mathsf{vk}, \mathbb{X}, \pi) \to \{0, 1\}$: The verification algorithm takes as input the verification key vk, the instance $\mathbb{X}$ and proof π, and outputs a bit.

We require the argument of knowledge to further satisfy the following list of properties:

Completeness: Completeness requires that for all relation $\mathcal{R}$, 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: Informally, knowledge soundness states that whenever a prover convinces the verifier of some instance $\mathbb{X}$ in the relation $\mathcal{R}$, the prover must also know an explicit witness w such that $\mathcal{R}(\mathbb{X}, w) = 1$. More formally, for any *efficient* adversary A, there must exist an *efficient* extractor $\mathcal{E}$ such that:

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

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

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

Succinct, Non-interactive Argument of Knowledge (SNARK) We further say that the argument is succinct and non-interactive if both the proof size $|\pi|$ and the running them of Verify are sublinear in the size of the instance-witness pair $(|\mathbb{X}| + |w|)$.

We are especially interested in the following special class of SNARKs which builds on top of vector commitment schemes:

Definition 2.2 (Commit-and-Prove SNARKs)**.** *A commit-and-prove SNARK associated with a vector commitment scheme* $(\mathsf{KeyGen}, \mathsf{Commit})$ *and a relation* $\mathcal{R}^*(,\cdot,)$ *is a succinct non-interactive argument of knowledge* $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ *for the following relation,* $\mathcal{R}_{\mathsf{ck}}((\mathbb{X}, c), w) = 1 \iff \mathcal{R}^*(\mathbb{X}, w) = 1 \wedge c = \mathsf{Commit}(\mathsf{ck}, w)$*, where* $\mathsf{ck} \leftarrow \mathsf{KeyGen}(1^\lambda)$ *is the commitment key.*

2.3.1 A Brief Survey of SNARK Field Choices

Most existing commit-and-prove SNARKs are inherently designed to support proving arithmetic relations over certain field. Nonetheless, depending on the underlying vector commitment schemes, the field choices can vary a lot. Here we give a brief survey about different vector commitment schemes and their corresponding field choices. Readers may also consult the survey of [Tha23], Chapter 19.3 for more details.

- The first category utilizes a commitment scheme that is based on certain algebraic hardness in a known-order group. The examples of such are [KZG10] [BBB$^+$18]. These schemes are widely adopted in many blockchain applications due to their extremely small commitment size. However, the choice of field is very limited due to many required properties of the group. Popular choices are elliptic curve groups BLS12-381 and BN-254, where the group order p^* is a fixed $\approx$ **255**-bit prime. The corresponding field is naturally $\mathbb{F}_{p^*}$.
- The second category utilizes a commitment scheme that is based on collision-resistant hash functions. The examples of such are [COS20][ZXZS20]. Here the field choice is rather flexible but ideally needs to support efficient FFT operations. Very often we choose $\mathbb{F}_p^*$ such that p^* is $\approx$ **64**-bit prime number.
- The third category utilizes hardness of unknown order groups [BFS20, CFKS22, AGL$^+$23, SB23]. These systems are only of theoretical interests due to slow running time hence out of our scope.

In this paper we mainly consider the first category as they are among the most popular SNARKs today. Nevertheless, due to the prime p^* (hence order of elliptic curve) being very large, each group operation is very slow. For this reason the prover's efficiency in these systems is often determined solely by the number of group operations that she needs to perform.

2.4 Existentially Quantified Circuits

Existentially quantified circuits (EQCs) [OBW22] are circuits which consist of sets of wires taking values from some domain (such as the prime field $\mathbb{F}_p$) and constraints that express certain relationships among wire values (such as the constraint $x \cdot y = z$). EQCs have two kinds of wire values: explicit inputs values which are assigned to input wire values at the start of execution, and existentially quantified wire values, which may take any value consistent with the explicit input values and the set of constraints.

In this work we are mainly interested in a family of "arithmetic EQCs" where the constraints are over the arithmetic operators $(*/\cdot, +, -)$ corresponding to multiplication/addition/subtraction in some given field.

2.5 Rank-1 Constraint Systems

Definition 2.3 (R1CS)**.** *Rank-1 Constraint Systems is a type of commonly used arithmetic EQC in cryptographic proof systems. An* R1CS *instance consists of a tuple* $\mathbb{X} = (\mathbb{F}, A, B, C, io, m, n)$ *where io denotes the public input and output of the instance, and three matrices* $A, B, C \in \mathbb{F}^{m \times (1+n)}$ *with* $n \geq |io|$*.*

An R1CS *instance is said to be* satisfiable *if there exists a witness* $w \in \mathbb{F}^{n-|io|}$ *such that* $(A \cdot z) \circ (B \cdot z) = (C \cdot z)$*, where* $z = (1, io, w) \in \mathbb{F}^{n+1}$ *(sometimes called* extended witness*),* $\cdot$ *is the matrix-vector product*

and $\circ$ is the Hadamard (entry-wise) product. We denote the satisfiable condition by $\mathcal{R}_{\mathsf{R1CS}}(\mathbb{X}, w) = 1$.

In this work we introduce another useful property whose importance will later become clear: We say that an R1CS *instance is k-**bounded** if for every witness w that makes* R1CS satisfiable*, each entry of the following vectors (matrices) $(z,\ A,\ B,\ C,\ A \cdot z,\ B \cdot z,\ C \cdot z)$ is within $[0, k]$ (here one must consider the operation $(A \cdot z) \circ (B \cdot z) = (C \cdot z)$ over the integers, not over field $\mathbb{F}$).*

2.5.1 EQC Compiler

An EQC compiler is a compiler infrastructure which takes a computer program as input, and produces an EQC instance such that the resulting EQC instance encodes the satisfiability of the program. In particular, consider the EQC compiler that outputs an R1CS instance:

Definition 2.4 (R1CS-type EQC compiler)**.** *Let P be a computer program which takes some value x as input, and let y be the alleged output. A* R1CS*-type EQC compiler takes as input the program P, its input/output values (x, y) and produces an* R1CS *instance $\mathbb{X} = (\mathbb{F}, A, B, C, io = x||y, m, n)$. The EQC compiler must satisfy completeness and soundness as follows: The instance $\mathbb{X}$ is satisfiable if and only if $P(x) = y$.*

We consider two additional properties of EQC compiler which meet our interests:

- **k-bounded**: The compiler always output a **k-bounded** R1CS instance.
- **p-satisfiable**: The compiler output a R1CS instance over certain prime field $\mathbb{F}_p$ (i.e. $\mathbb{X} = (\mathbb{F}_p, A, B, C, \dots)$). Furthermore, for any prime $p' \geq p$, the field $\mathbb{F}_{p'}$ can be used to substitute $\mathbb{F}_p$ in the sense that the instance $\mathbb{X}$ is satisfiable over $\mathbb{F}_p$ if and only if it is satisfiable over $\mathbb{F}_{p'}$.

We observe that most EQC compilers [OBW22] [KPS18] [cir] admit those properties by design for certain values of k and p. Furthermore, for computer programs where all the variables are **sufficiently small**, such as AES, SHA2 and SHA3, these EQC compilers are naturally k-bounded and p-satisfiable for **small** values of k and p (for example, in AES and SHA2-256, [cir] admits $k = p = 2^8$). For all other programs, these compilers can still be equipped with small values of k and p by allowing to output larger EQC instances.

SNARK Backend and Frontend In the modern language of SNARKs, the process of compiling a computer program into a suitable EQC instance (such as R1CS) is often referred to as the SNARK **frontend**. These instances are then consumed by various SNARKs schemes targeting for circuit satisfiability. Such process of proving EQC instance is often referred to as the SNARK **backend**.

2.6 Highly Repetitive Computation

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

2.6.1 Data Parallel (SIMD) Computation

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

As a concrete example, consider the SIMD computation C where the sub-computation G is repeated ℓ times with ℓ different independent inputs $(x_1, \dots, x_\ell)$. That is: $C(x_1, \dots, x_\ell) = (G(x_1), \dots, G(x_\ell))$.

2.6.2 Incremental Computation

Incremental computation captures most recursions (and while loops) in the program: It involves a fixed sub-computation being applied to multiple pieces of dependent inputs.

As a concrete example, consider the incremental computation C where the sub-computation G is repeated ℓ times iteratively: That is: $C(x) = \underbrace{G(G \ldots G(x))}_{\ell \text{ times}}$.

3 Roadmap

The following sections are organized as follows: In section 4, we release the initial blueprint of our improved frontend by introducing our first core technique: **CRT Packing** for SIMD circuits. Naively applying such packing technique indeed yields an "improved frontend", but it isn't directly compatible with most backends due to a mismatch between the frontend ring and backend field. Therefore we introduce in section 5 our second core technique: **Fast Ring Emulation.** It allows prover to efficiently emulate any 'non-native' ring arithmetics inside any fixed, large prime field. In section 6 we show how to use those compilers to build a more prover efficient, commit-and-prove SNARK for SIMD computations. Finally we provide detailed implementation and evaluation in section 8.

4 First Technique: CRT Packing

In this section we build on an **information theoretic** technique called **CRT packing** and show how to apply this technique so as to design a better frontend compiler for any computation that is *data-parallel* (SIMD). We first illustrate using a toy example, and then move to the general case. The resulting frontend, although drastically improved over naive frontend, raises a concern of backend-compatibility as we will discuss lastly.

4.1 CRT Packing

Intuitively, our starting point is to develop an efficient, information theoretic mechanism which allows to 'pack' several small, distinct prime fields into one big arithmetic ring, such that those individual field arithmetic behavior can all be explained by single behavior of the final ring. Thankfully, CRT provides exactly such mechanism we're looking for.

4.2 A Toy Example

Consider the following SIMD computation C which computes the XOR of two length ℓ bit strings: $\mathbf{x} \in \{0,1\}^\ell,\ \mathbf{y} \in \{0,1\}^\ell : \ C(\mathbf{x}, \mathbf{y}) \to \mathbf{z} = \mathbf{x} \oplus \mathbf{y} \in \{0,1\}^\ell$. We can view the computation C as ℓ parallel executions of the sub-computation G which outputs two-bitwise XOR: $x_i \in \{0,1\},\ y_i \in \{0,1\} : G(x_i, y_i) \to z_i = x_i \oplus y_i \in \{0,1\}$. In order to check the correctness of computation C, we could naively check all $G(x_i, y_i)$'s are computed correctly. More specifically, we build an arithmetic EQC for each sub-computation G_i on input wires (x_i, y_i): It first enforces all wires to be binary (i.e. they are all in $\{0,1\}$), and then it checks that the output wire is the XOR of input wires. This circuit representation can be arithmetically described by the following wire constraints:

- $x_i^2 - x_i = 0$; (Which enforces x_i to be binary.)
- $y_i^2 - y_i = 0;\ z_i^2 - z_i = 0;$
- $x_i + y_i - 2x_i \cdot y_i = z_i$. (Which enforces $z_i = x_i \oplus y_i$.)

Notice that these constraints are implicitly defined over the whole integer ring. Nonetheless, observe that it is also sufficient to ask these constraints to hold over any prime field $\mathbb{F}_p$ where $p \geq 2$. This is because the first three constraints implicitly ensure that all of $x_1,\ y_1,\ z_1 \leq 1$ whenever $p \geq 2$. Therefore $x_1 + y_1 - 2x_1 \cdot y_1 \leq 2 \leq p$. In other words, if the fourth constraint holds over $\mathbb{F}_p$, then it already holds over the integers as there are no 'wrap around' happening over $\mathbb{F}_p$.

In order to express the correctness of all copies of G_i into some EQC, one can simply repeat the above wiring constraints for each copy $G_i(x_i, y_i)$. This naive approach yields an EQC consisting of 4ℓ wire constraints defined over $\mathbb{F}_p$.

Can we do better than this? Indeed, recall that if we have ℓ different prime fields and they all perform the same arithmetic operation, then CRT allows us to reduce them into just one single arithmetic operation. This suggests a natural way to pack all the ℓ EQCs together. For each $i \in [\ell]$, we will pick a different prime number (for example, we pick $q_1 = 2$, $q_2 = 3$, and so on). Then we 'lift' the i^{th} EQC into its respective field by defining the i^{th} set of wiring constraints over the prime field $\mathbb{F}_{q_i}$. Importantly, since each $q_i \geq 2$, the lifted i^{th} EQC still represents the correct execution of G_i.

Let q be the product of those ℓ primes. We are ready to 'pack' all of ℓ EQCs together: firstly we pack their the input/outputs: Let's use the interface $\mathsf{CRT.Pack}(x_1, \ldots, x_\ell) \to x \in \mathbb{Z}_q$ to denote the CRT packing for all the x_is, and similarly for y_is $\to y$ and z_is $\to z$. Now consider the following 'packed' EQC with respect to three 'packed' input/output values defined over the 'packed' ring $\mathbb{Z}_q$. Due to CRT isomorphism, if these packed wire values $(x, y, z \in \mathbb{Z}_q)$ satisfy the above EQC, then for each $i \in [\ell]$, we must have those unpacked values $(x_i = x \mod q_i,\ y_i = y \mod q_i,\ z_i = z \mod q_i)$ satisfying the same EQC defined over individual prime field $\mathbb{F}_{q_i}$. That is to say, it must hold that $G(x_i, y_i) = z_i$ for all i.

To briefly summarize what we have done so far, compared to the naive method of constructing ℓ EQCs, we packed all of them into one single EQC at the expense of expanding the field (in fact, the ring) where this EQC is defined over. Due to CRT isomorphism, the packed EQC is information theoretically equivalent to having all previous ℓ EQCs. Nonetheless it enjoys $\ell\times$ less wiring constraints.

4.3 Handling General SIMD Computations

We now design a frontend compiler which can pack general SIMD computations. Let's consider any SIMD computation $C(x_1, \ldots, x_\ell) = (G(x_1), \ldots, G(x_\ell))$.

Again, we first represent those subcomputations in their EQCs. For ease of implementation and explaining, from now on we only work with R1CS-type EQC. Let's apply an R1CS-type EQC compiler on G, which outputs some instance $\mathbb{X}_G = (\mathbb{F}_p, A_G, B_G, C_G, \text{io} = \bot, m, n)$. For each sub-computation $i \in [\ell]$ with input x_i, let y_i be its alleged output. We denote by $\mathbb{X}_{G^i}$ where we substitute the empty input/output values io with the corresponding value (x_i, y_i): That is, $\text{io}_i = x_i || y_i$. It follows that $\mathbb{X}_{G^i}$ encodes the correctness (or satisfiability) of the sub-computation G over input value x_i and output y_i.

In order to enable 'lifting' each instance $\mathbb{X}_{G^i}$ into a separate prime field, we additionally require the above compiler to be **p-satisfiable**[2]. This guarantees that 'lifting' won't hurt the encoded correctness/satisfiability condition of R1CS so long as the lifted prime field is larger than $\mathbb{F}_p$.

Now let w_i be prover's alleged witness for the i^{th} R1CS instance $\mathbb{X}_{G^i}$. Let $\mathbf{z_i} = (1, \text{io}_i, w_i) \in \mathbb{Z}_k^{n+1}$ be its extended witness vector.

We want to use CRT packing to pack all instances $\{\mathbb{X}_{G^i}\}_{i\in[\ell]}$ into just one instance. To achieve this, first choose ℓ smallest different prime numbers $(q_1, \ldots, q_\ell)$ such that each $q_i \geq p$, and let $q = \prod_{i=1}^{\ell} q_i$. Then pack the input/outputs as $\text{io}_C = \mathsf{CRT.Pack}(\text{io}_1, \ldots, \text{io}_\ell)$, and similarly pack all (extended) witness into $(w_C, \mathbf{z_C})$. Notice that $\mathbf{z_C} = (1, \text{io}_C, w_C)$. Finally define the packed instance as $\mathbb{X}_C = (\mathbb{Z}_q, A_G, B_G, C_G, \text{io}_C, m, n)$.

Observe that due to CRT isomorphism, we have:

$$\exists\ w_C : (A_G \cdot \mathbf{z_C}) \circ (B_G \cdot \mathbf{z_C}) = C_G \cdot \mathbf{z_C} \text{ over } \mathbb{F}_q$$
$$\text{if and only if}$$
$$\forall i \in [\ell],\ \exists\ w_i : (A_G \cdot \mathbf{z_i}) \circ (B_G \cdot \mathbf{z_i}) = C_G \cdot \mathbf{z_i} \text{ over } \mathbb{F}_{q_i}.$$

Moreover, due to p-satisfiability, this implies the second part also holds over $\mathbb{F}_p$. In other words, the packed instance $\mathbb{X}_C$ is satisfiable if and only if all instances $\{\mathbb{X}_{G^i}\}_{i\in[\ell]}$ are satisfiable. As a result, $\mathbb{X}_C$ indeed encodes the satisfiability of the whole SIMD computation C.

Acute readers may already observe that as the number of packed EQC grows, the size of ring $\mathbb{Z}_q$ also increases. Jumping ahead, due to some size restriction of $\mathbb{Z}_q$, we cannot continue packing at some point.

[2] As we will discuss in section 5, this compiler also needs to be **k-bounded**.

For this reason we will first fix a maximum quantity for packing, and from now on we refer to this as the packing factor, denoted by ℓ. For general SIMD computation C consisting of N total copies of G, we will split them into N/ℓ batches, and apply packing to each batch.

In figure 1 we formalize the above packing technique into a frontend compiler. We slightly abuse our previous notation and still denote it by $\mathsf{CRT.Pack}$. It takes as input $\ell \times \mathsf{R1CS}$ instances, where each instance is assumed to be outputted by some *k-bounded* and *p-satisfiable* $\mathsf{R1CS}$-type EQC compiler. It outputs one single packed $\mathsf{R1CS}$ instance.

$\underline{\mathsf{CRT.Pack}(\mathbb{X}_{G^1}, \ldots, \mathbb{X}_{G^\ell})}$

1. For $i \in [\ell]$, parse $\mathbb{X}_{G^i} = (\mathbb{F}_p, A_G, B_G, C_G, \mathrm{io}^i, m, n)$.
2. Choose ℓ smallest different prime numbers $(q_1, \ldots, q_\ell)$ such that each $q_i \geq p$. Let $q = \prod_{i=1}^{\ell} q_i$.
3. Pack the input/output values $\{\mathrm{io}^i\}_{i\in[\ell]}$ via CRT packing: $\mathsf{CRT.Pack}(\{\mathrm{io}^i\}_{i\in[\ell]}) \to \mathrm{io}$.
4. Output $\mathbb{X}_C^q = (\mathbb{Z}_q, A_G, B_G, C_G, \mathrm{io}, m, n)$.

Figure 1: $\mathsf{CRT.Pack}$ Compiler for SIMD Instances

4.4 Ensuring Backend Compatibility

The aforementioned frontend compiler for SIMD almost yields an 'improved' frontend. More specifically, for any SIMD computation, the compiler yields an $\mathsf{R1CS}$ instance whose size is ℓ times smaller than that of the naive method.

The only caveat now is that the resulting frontend instance $\mathbb{X}_C$ is defined over some specifically chosen ring $\mathbb{Z}_q$, whereas most backends only works with a **fixed** prime field $\mathbb{F}_{p^*}$. As a result, despite we have an improved instance $\mathbb{X}_C$ after CRT packing, we don't have any SNARK backends which can be used to prove the satisfiability of such instance. That is to say, we now face a backend compatibility issue.

In fact, the difficulty we're facing is to prove arithmetic operations that are 'non-native' to the backend field. This turns out to be one of major annoying issues in SNARKs. Although some existing solutions have been proposed to deal with this, none of them is efficient. Some utilize bit decomposition, such as [KPS18], but this introduces a large number of constraints, thus jeopardize efficiency. Others utilize certain special ring encodings, such as [GNSV21], but has to give up an desirable SNARK feature called public verifiablity and still poses many constraints onto the backend. The bottom line is, using any of those solutions, we will immediately give up all the efficiency gain we've achieved through packing.

5 Second Technique: Fast Ring Emulation

Our second contribution is to propose an efficient solution to above problem. We call such technique **Fast Ring Emulation**. As the name suggests, it allows the prover to quickly 'emulate' the arithmetic behaviors of some non-native ring, such as $\mathbb{Z}_q$ in our example, inside a fixed prime field $\mathbb{F}_{p^*}$. The core of such technique is an **information theoretic**, "degree-2 homomorphic" embedding scheme: It allows the prover to embed an element of $\mathbb{Z}_q$ inside a (potentially non-unique) element of $\mathbb{Z}_{p^*}$. Furthermore, due to a degree-2 homomorphism, the prover can efficiently emulate any degree-2, $\mathbb{Z}_q$ arithmetic operations which involve any finite number of additions/constant multiplications, or **one single** multiplication in $\mathbb{Z}_q$, so long as size restriction $q << \sqrt{p^*}$ holds. Crucially, the aforementioned operations are considered to be '$\mathsf{R1CS}$-complete' since such EQC is exactly degree-2.

5.1 A Toy Example

Let's again start with a toy example: Suppose the prover holds three witness values (a, b, c) and wants to prove some constraint relation $a \cdot b = c$ over $\mathbb{Z}_q$. This corresponds to the modulo arithmetic relation $a \cdot b = c \mod q$. On the other hand, the native arithmetics are over $\mathbb{F}_{p^*}$, hence enforcing above constraint implicitly $\mod p^*$. Readers may already notice that even if $a \cdot b = c$ holds over $\mathbb{Z}_q$, it does not necessarily imply that $a \cdot b = c \mod p^*$. This becomes an issue even for an honest prover! In order to help honest

provers, let's allow the prover to supply an additional shift value k and instead prove that $a \cdot b = c + k \cdot q \mod p^*$. Notice that honest prover can always find shift k such that $a \cdot b = c + k \cdot q$ is true over the integers, hence also true over $\mathbb{F}_{p^*}$. Nonetheless, a dishonest prover may pick a 'cheating' shift value k' such that even if $a \cdot b \neq c \mod q$, the relation $a \cdot b = c + k' \cdot q \mod p^*$ still holds!

To circumvent all those problems, we will in fact use a different representation of $\mathbb{F}_{p^*}$ elements, called rational representatives. This notion has been used in early literature [FS01] and more recently leveraged to build efficient range proofs [CKLR21]. Informally, we say that an element $a \in \mathbb{F}_{p^*}$ can be represented by a rational $\frac{a_1}{a_2}$ if it holds that $a = \frac{a_1}{a_2} \mod p^*$. For the sake of simplicity let's assume for now that p^* must be a prime so that this relation is always well-defined. We defer all related concepts about rational representatives to section 5.2.

Recall that prover has witness values (a, b, c, k). Since the native field is $\mathbb{F}_{p^*}$, we must consider all of them to be $\mathbb{F}_{p^*}$ elements. Let's see what happens when substituting those elements with their rational representatives in the previous constraint relation:

$$\frac{a_1}{a_2} \cdot \frac{b_1}{b_2} = \frac{c_1}{c_2} + \frac{k_1}{k_2} \cdot q \mod p^*,$$

Now observe the following: If all these rational representatives admit small numerators and small denominators (e.g. $(a_1, a_2, b_1, \dots)$ are relatively small), then this arithmetic relation indeed holds over the field of rational numbers, instead of just holding over modulo p^*. To see this, notice that we can first multiply both the left and right hand sides by the LCM of denominators, which yields:

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

Since each individual variables are assumed to be small, the product of them should still be small. As long we we have $\max(a_1 \cdot b_1 \cdot c_2 \cdot k_2,\ c_1 \cdot a_2 \cdot b_2 \cdot k_2 + k_1 \cdot a_2 \cdot b_2 \cdot c_2 \cdot q) < p^*$, then in fact the above equation holds over the integers!

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

At this point we can divide back the previous LCM, and get

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

The key take away so far is the following: whenever the prover utilizes rational representatives that have small numerators and denominators, then she can completely get rid of the dependence on the field $\mathbb{F}_{p^*}$. Whatever relations the prover manages to prove using those representatives, those relations must hold unconditionally over the rational numbers. The same observation has also been made in [CGKR22, GJJZ22] for the purpose of proving relations over rational numbers.

In this work, we take one step further: On top of viewing them as rational relations, we again treat those rational numbers as elements from a 'different' ring. Since the eventual goal of prover is to prove relations over $\mathbb{Z}_q$, let's conceptually view these rational numbers again as rational representatives, nonetheless of $\mathbb{Z}_q$ elements instead of $\mathbb{Z}_{p^*}$ elements. This is equivalent to taking this equation over modulo q. Let's assume for now that q is also a prime for simplicity. Concretely, let $a' = \frac{a_1}{a_2} \mod q$ (similarly for b, c, k), then it holds that:

$$a' = b' \cdot c' \mod q,$$

That is to say, the prover conceptually manages to prove the relation $a' = b' \cdot c'$ over $\mathbb{Z}_q$, despite that her witness values (a, b, c, k) are all in fact elements from $\mathbb{F}_{p^*}$.

To summarize, whenever the prover utilizes $\mathbb{F}_{p^*}$ elements (a, b, c, k) such that they admit rational representatives where the denominators and numerators are sufficiently small (we will call such representatives bounded rational representatives), then in fact we can view these elements as embedded $\mathbb{Z}_q$ elements. Furthermore, this embedding is "somewhat homomorphic": Arithmetic operations over $\mathbb{F}_{p^*}$ elements correspond to the same operations over their embedded $\mathbb{Z}_q$ elements. This homomorphism thus allows the prover to emulate the behavior of non-native ring $\mathbb{Z}_q$ inside $\mathbb{F}_{p^*}$. However, as we will see later,

multiplicative homomorphism comes at the expense of ensuring a sufficient gap between q and p^*. In order to maximize our packing factor, we will only allow one multiplication to be emulated, which is still sufficient for degree-2 constraint system like R1CS.

Importantly, honest provers can safely use $\mathbb{Z}_q$ elements to prove this relation since any element $a \in \mathbb{Z}_q$ can be trivially written as its representative $\frac{a}{1}$. Since we assume $a < q << p^*$, this representative is indeed a bounded rational, thus allowed to be used. In other words, such embedding creates no overheads for honest provers.

Enforcing Bounded Rationals with Batch-PoSO Recall that in order for aforementioned embedding technique to work, we need to ensure that the prover's witness values only consist of $\mathbb{F}_{p^*}$ elements which can be represented by bounded rationals. For example, when working with $\mathbb{F}_7$, if we demand rationals with numerator and denominator both be bounded by $[-1, 1]$, then the element $(2, 3, 4, 5)$ can not be represented by such rationals. To enforce bounded rationals, we rely on a recent **information theoretic** technique called Batch Proof-of-Short-Opening (batch-PoSO) [CGKR22, GJJZ22]. Informally, let $\mathbf{w} \in \mathbb{F}_{p^*}^n$ be prover's witness vector. Batch-PoSO uses the following test procedure: Sample a short random vector $\mathbf{r}$ where each entry is small. Then check if the inner product $< \mathbf{r}, \mathbf{w} >$ is a small value. Intuitively, since an honest prover will only use 'trivial' rational representative with small numerator and $'1'$ being the denominator, this inner product should remain small. On the other hand, if the inner product is small with high probability, then by averaging argument, for each index $i \in [n]$, there must exist two short vectors that only differ at the i^{th} index such that their inner products with $\mathbf{z}$ are both small. Thus one can extract each $\mathbf{z}[i]$ as a bounded fraction by taking the difference. We defer the details and security proofs of this protocol to section 5.3.

Enforcing Well-defined Rationals Lastly, since our CRT packing mandates q to be a composite number, the rational representatives of $\mathbb{Z}_q$ are not all well-defined due to lack of inverse. To prevent a dishonest prover from taking such advantage, we make another simple observation: So long as the denominator is sufficiently bounded such that it's less than the smallest divisor of q, then the corresponding rational must be well-defined. Therefore, Batch-PoSO can also be used to enforce well-defined rationals. We defer the details to lemma 5.4.

5.2 Related Concepts of Rational Representative

Consider the prime field $\mathbb{F}_p$. Other than the usual way to represent its elements: $[0, p-1]$, one can also represent them as a set of rational numbers. We take the following definition of rational representatives from [CKLR21, CGKR22]:

Definition 5.1 (Rational Representative)**.** *Let $\mathbb{Q}$ be the set of rationals (we always assume the numerator and denominator are coprime), that is:*

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

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

Note that for each $x \in \mathbb{F}_p$, it can have multiple rational representatives.

One can also generalize this notion to any quotient ring $\mathbb{Z}_q$, where q is not necessarily a prime number. In this case we must restrict ourselves to the set of rational numbers whose denominators are invertible modulo q, thus being well-defined.

Definition 5.2 ($q-$Invertible Rational Representative)**.** *Let $\mathbb{Q}_q$ be the following set of rationals:*

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

Then for any value $x \in \mathbb{Z}_q$, we say that x is represented by the rational $\frac{n}{d} \in \mathbb{Q}_q$ if $x = n \cdot d^{-1} \mod q$.

Another useful set of rational representatives, as hinted in the previous section, are the set of rationals with small denominators and numerators:

Definition 5.3 (Bounded Rational Representative)**.** *The set of bounded rational $\mathbb{Q}_{N,D} \subseteq \mathbb{Q}$ contains all the rationals whose numerator is bounded by N and denominator bounded by D, that is:*

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

Bounded Representative Is All You Need We make a simple observation of the following relationship between the set Q_q and $\mathbb{Q}_{N,D}$:

Lemma 5.4 (Criterion for q-invertible)**.** *Let $q_{\min}$ be the smallest divisor of q. If $D < q_{\min}$, then all rationals in $\mathbb{Q}_{N,D}$ are also q-invertible. In other words, we have $\mathbb{Q}_{N,D} \subseteq \mathbb{Q}_q$.*

Proof. Since any denominator smaller than $q_{\min}$ must be coprime to q, the corresponding rational number must be q-invertible. Jumping ahead, in our setting we only consider q whose smallest divisor is still "relatively large". □

Now we slightly abuse the notation: For $x \in \mathbb{F}_{p^*}$, we denote by $x \in \mathbb{Q}_{N,D}$ if there is a representative in $\mathbb{Q}_{N,D}$ of x. Recall that in our application we want to enforce the prover to only use a set of $\mathbb{F}_{p^*}$ elements such that their representatives are bounded. Furthermore, those elements are given as a vector commitment. In general, we ask the following question: For any choice of (N, D), and for any vector of elements $\mathbf{z} \in \mathbb{F}_{p^*}^n$ given as its commitment: $c \leftarrow \mathsf{Commit}(\mathsf{ck}, \mathbf{z})$, how can we design a test which enforces that this committed vector satisfies $\mathbf{z} \subseteq \mathbb{Q}_{N,D}$?

5.3 Proof of Short Opening (PoSO)

In [CKLR21] and subsequent works [CGKR22, GJJZ22], the authors provide a solution to the above question with a one-round commit-and-prove protocol called Batch Proof of Short Opening (Batch-PoSO). The name arises from the fact that the proof shows the existence of some 'short' (bounded) rational representative that can be used to open the committed vector, hence short opening. More formally, it is a commit-and-prove protocol for the following gap language $(L_{R,1}, L_{N,D})$:

- A vector $\mathbf{z} \in \mathbb{Z}_{p^*}^n$ is said to be in the language $L_{R,1}$ if $\mathbf{z} \subseteq Q_{R,1}$.
- A vector $\mathbf{z} \in \mathbb{Z}_{p^*}^n$ is said to be in the language $L_{N,D}$ if $\mathbf{z} \subseteq Q_{N,D}$.

In the setting of gap language, we require the verifier to always accept all instance in $L_{R,1}$, and reject all instance not in $L_{N,D}$ with high probability. Intuitively, when the ring to be emulated is chosen to be $\mathbb{Z}_q$, we will just set $R = q << N$. This ensures that all elements of emulated ring belong to the accepting instance, which helps an honest prover to always succeed in the protocol.

Overview Before presenting the details of batch-PoSO, we first present a high level overview. It is an one-round interactive protocol which builds on top of any commit-and-prove SNARK as follows: Let $c = \mathsf{Commit}(\mathbf{z})$ be the committed vector which the prover first sends the verifier. Upon seeing the commitment, verifier will send a short random vector $\mathbf{r} \in \mathbb{Z}_{p^*}^n$ such that each entry $\mathbf{r}[i]$ is small (i.e. $\forall i \in [n], \mathbf{r}[i] \in [0, D)$). The prover then uses the commit-and-prove SNARK to produce a proof attesting the statement that $v \coloneqq < \mathbf{z} \cdot \mathbf{r} >$ over $\mathbb{F}_{p^*}$, with respect to the committed witness $\mathbf{z}$. Notice that this relation is over the native field $\mathbb{F}_{p^*}$, hence it can be directly proved. The verifier then accepts if SNARK proof is valid and also $v \in [N]$. The formal description of Batch-PoSO protocol is provided in figure 2.

5.3.1 Security proof for Batch-PoSO

We prove that the Batch-PoSO protocol described in figure 2 satisfies completeness and soundness via the following two claims:

Claim 5.5 (Completeness of Batch-PoSO)**.** *Whenever $N \geq R \cdot D \cdot n$, the above Batch-PoSO satisfies completeness.*

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

Batch-PoSO: A Commit-and-Prove Construction

- **Ingredients:** Let $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ be a commit-and-prove SNARK with respect to some vector commitment scheme $(\mathsf{KeyGen}, \mathsf{Commit})$ with native field $\mathbb{F}_{p^*}$.
- **Instance and Language:** The prover holds some vector $\mathbf{z} \in \mathbb{Z}_{p^*}^n$ and its commitment $c = \mathsf{Commit}(\mathsf{ck}, \mathbf{z})$. The gap language $(L_{R,1}, L_{N,D})$ is parametrized by (R, N, D).
- **Protocol Description:**
  1. The prover sends commitment c to the verifier.
  2. The verifier samples $\mathbf{r} \in \mathbb{Z}_{p^*}^n$ such that each entry $\mathbf{r}[i]$ is small (i.e. $\forall i \in [n], \mathbf{r}[i] \in [0, D)$), and then sends them to the prover.
  3. The prover and verifier define the relation $\mathcal{R}^*((\mathbf{r}, v), \mathbf{z}) = 1 \iff v :=< \mathbf{z} \cdot \mathbf{r} >$, and then prove the instance $(\mathbf{r}, v)$ in this relation as follows:
     - Prover and verifier convert the above relation into following $\mathsf{R1CS}$ relation: $\mathbb{X}_{\mathsf{PoSO}} = (\mathbb{F}_{p^*}, A, B, C, \text{io} = [1, v], 1, n + 1)$, where $A = [\mathbf{r}[1], \ldots, \mathbf{r}[n], 0, 0]$, $B = [0, \ldots, 0, 1]$ and $C = [0, \ldots, 1, 0]$.
     - The prover uses the commit-and-prove SNARK to produce a proof π attesting that $\mathcal{R}_{\mathsf{R1CS}}(\mathbb{X}_{\mathsf{PoSO}}, \mathbf{z}) = 1 \wedge c = \mathsf{Commit}(\mathsf{ck}, \mathbf{z})$.
  4. The verifier checks that π is a valid proof and $v \in [N]$. If so, the verifier accepts, otherwise it rejects.

Figure 2: Description of Batch-PoSO.

Claim 5.6 (Soundness of Batch-PoSO)**.** *For any fixed constant N, and for any $\{z_i\}_{i\in[n]}$ with $z_i \in \mathbb{F}_{p^*}$ for each i, if*

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

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

As a consequence of this claim, the soundness error of Batch-PoSO is at most $1/D$ plus the soundness error of the underlying commit-and-prove SNARK.

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

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

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

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

Since there are only D choices of r_i, there exists $r_i^1, r_i^2 \in [0, D), (r_i^1 > r_i^2)$ such that

$$\sum_{j\neq i}^{n} r_j^* \cdot z_j + r_i^1 \cdot z_i \in [N] \bigwedge \sum_{j\neq i}^{n} r_j^* \cdot z_j + r_i^2 \cdot z_i \in [N].$$

Now we set $z_{i,2} \coloneqq r_i^1 - r_i^2 \in [1, D)$, and set

$$z_{i,1} \coloneqq \left(\sum_{j\neq i}^{n} r_j^* \cdot z_j + r_i^1 \cdot z_i\right) - \left(\sum_{j\neq i}^{n} r_j^* \cdot z_j + r_i^2 \cdot z_i\right)$$

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

Due to one technicality of our frontend, we also prove a stronger statement regarding the least common multiple of all the denominators of those rationals in appendix section 9.1.

5.4 FRE: Bringing Fast Ring Emulation into Frontend

In this section we show how to incorporate fast ring emulation technique into any frontends involving arithmetic in some non-native field. In particular, let $\mathbb{X}^q$ be some R1CS instance defined over some non-native ring $\mathbb{Z}_q$. We design a compiler such that it compiles $\mathbb{X}^q$ into another instance $\mathbb{X}^{p^*}$ which is defined over the native field $\mathbb{F}_{p^*}$. Importantly, we require that $\mathbb{X}^{p^*}$ inherits the same satisfiability condition as $\mathbb{X}^q$ with high probability. We call such compiler Fast Ring Emulator (FRE).

Workflow At a high level, FRE is a two-stage compiler that is designed to work with any commit-and-prove SNARKs. The first stage of FRE is denoted by FRE.Fit: It takes as input some 'non-native' instance $\mathbb{X}^q$, and outputs a native instance $\mathbb{X}^{p^*}_{\mathsf{Fit}}$. This is done by 'fitting' each constraint over $\mathbb{Z}_q$ into another constraint over $\mathbb{F}_{p^*}$. As a result, if $\mathbb{X}^q$ is satisfiable, then $\mathbb{X}^{p^*}_{\mathsf{Fit}}$ must also be satisfiable. Nonetheless, the reverse implication does not necessarily hold.

After applying the first stage of FRE, the prover will compute all witness values for $\mathbb{X}^{p^*}_{\mathsf{Fit}}$, and then commit to its extended witness using the associated vector commitment scheme. Upon receiving the commitment, we will proceed to the second stage of FRE, which we denote by FRE.Emulate: It takes as input the previous instance $\mathbb{X}^{p^*}_{\mathsf{Fit}}$ and outputs a final instance $\mathbb{X}^{p^*}_{\mathsf{Emulate}}$. This instance shares the same witness value as that being previously committed. Importantly, we have the guarantee that $\mathbb{X}^{p^*}_{\mathsf{Emulate}}$ is satisfiable over $\mathbb{F}_{p^*}$ if and only if $\mathbb{X}^q$ is satisfiable over $\mathbb{Z}_q$, except with negligible probability. Now the prover can just use the commit-and-prove SNARK to show that $\mathbb{X}^{p^*}_{\mathsf{Emulate}}$ is satisfiable with respect to the committed witness.

We now give an overview of details in each stage.

Fitting Stage: In this stage, we will add a shift value to each constraint so as to help the honest provers proving those constraint relations. For example, recall that in the toy example, in order for honest prover to prove $a \cdot b = c \mod q$, we allow for a shift k and modify the constraint to be $a \cdot b = c + k \cdot q \mod p^*$. The full description of FRE.Fit is provided in figure 3.

$\underline{\mathsf{FRE.Fit}(\mathbb{X}^q)}$

1. Parse $\mathbb{X}^q = (\mathbb{Z}_q, A, B, C, \mathrm{io}, m, n)$.
2. Let $A' = A||\mathsf{diag}^m(0)$, $B' = B||\mathsf{diag}^m(0)$, $C' = C||\mathsf{diag}^m(q)$.
3. Output $\mathbb{X}^{p^*}_{\mathsf{Fit}} = (\mathbb{F}_{p^*}, A', B', C', \mathrm{io}, m, (n+m))$.

Figure 3: Fitting Stage of FRE

Emulating Stage: Intuitively, the goal of second stage is to ensure that the witness values being committed after the first stage are all valid embeddings of $\mathbb{Z}_q$ elements. As discussed in section 5.2, this is equivalent to enforcing all witness values to be some bounded rational representatives, which can be done via Batch-PoSO protocol. Therefore, we will augment the previous instance $\mathbb{X}^{p^*}_{\mathsf{Fit}}$ to incorporate all the new R1CS constraints involved in this protocol. This is achieved through the following steps:

1. First, we set the parameters for Batch-PoSO.
2. Then sample some short random vector as specified by the protocol.
3. Let $\mathbb{X}_{\mathsf{PoSO}}$ be the corresponding R1CS instance of Batch-PoSO, as described in figure 2.
4. Recall that at the end of Batch-PoSO, it is required that the verifier checks that the claimed result of inner product is small. This check will be incorporated into the final $\mathbb{X}_{\mathsf{PoSO}}$ instance in the form of range proof constraints. Notice that we only need *one single range proof.*
5. Now augment instance $\mathbb{X}^{p^*}_{\mathsf{Fit}}$ to incorporate $\mathbb{X}_{\mathsf{PoSO}}$.

Finally, since a single invocation of Batch-PoSO incurs statistical soundness error $1/D$, we will repeat those constraints $\lambda/\log(D)$ times to achieve negligible soundness error. The description of FRE.Emulate is provided in figure 4.

$\underline{\mathsf{FRE.Emulate}(\mathbb{X}^{p^*}_{\mathsf{Fit}})}$

1. Parse $\mathbb{X}^{p^*}_{\mathsf{Fit}} = (\mathbb{F}_{p^*}, A', B', C', \text{io}, m, (n+m))$.
2. *Set* PoSO *parameters:*
   Let $q_{\min}$ be the smallest divisor of q. Define the gap language $(L_{R,1}, L_{N,D})$, where $R = q$, $D = q_{\min} - 1$, $N = q \cdot (q_{\min} - 1) \cdot (m+n)$.
3. *Incorporate* $\lambda/\log(D)$ *Batch-*PoSO*:*
   (a) Sample $\mathbf{r} = \mathbf{r}_1 || \ldots || \mathbf{r}^{\lambda/\log(D)}$, where each $\mathbf{r}^j \leftarrow [0, D)^{(m+n)}$.
   (b) *Add* PoSO *Constraints:* Define instance $\mathbb{X}_{\mathsf{PoSO}} = (\mathbb{F}_{p^*} A, B, C, \text{io} = \perp, \lambda/\log(D), m+n)$, where $A = \begin{bmatrix} \mathbf{r}^1 ||0||0 \\ \ldots \\ \mathbf{r}^{\lambda/\log(p)} ||0||0 \end{bmatrix}$, $B = \begin{bmatrix} 0 \ldots ||0||1 \\ \ldots \\ 0 \ldots ||0||1 \end{bmatrix}$ and $C = \begin{bmatrix} 0 \ldots ||1||0 \\ \ldots \\ 0 \ldots ||1||0 \end{bmatrix}$.
   (c) *Add Range Proof Constraints:* Let $v^j :=< \mathbf{z} \cdot \mathbf{r}^j >$ for all $i \in [\lambda/\log(D)]$. To enforce $v^j < N$, we use bit-decomposition method, informally:
      i. For each entry v^j, we denote by $b_1, \ldots, b_{\log(N)}$ its bit-decomposition. Now add following constraints to $\mathbb{X}_{\mathsf{PoSO}}$:
      ii. Add the R1CS constraint that $b_i^2 - b_i = 0$ for each bit b_i, which enforces b_i take binary values.
      iii. Add the R1CS constraint that $v^j = \sum_{i=1}^{\log(N)} 2^i \cdot b_i$, which shows the correct bit-decomposition.
4. *Augment* $\mathbb{X}^{p^*}_{\mathsf{Fit}}$*:*
   Output $\mathbb{X}^{p^*}_{\mathsf{Emulate}}$ as the vertical matrix-wise concatenation of $\mathbb{X}^{p^*}_{\mathsf{Fit}}$ and $\mathbb{X}_{\mathsf{PoSO}}$.

Figure 4: Emulating Stage of FRE

6 More Efficient zk-SNARK for SIMD Computations

To recap our discussion so far, in section 4 and 5 we introduced two **information theoretic** frontend compilers:

- Packing compiler CRT.Pack: For any SIMD computations, it packs many copies of R1CS instances into one single instance. Nonetheless the packed instance is defined over some non-native ring $\mathbb{Z}_q$.
- Fast Ring Emulator (FRE.Fit, FRE.Emulate): It compiles any non-native instance $\mathbb{X}^q$ into a native instance $\mathbb{X}_{p^*}$. The compilation process is two-stage and works with any commit-and-prove SNARKs.

In this section we illustrate how to combine those two frontend compilers to build more efficient SNARK frontend for SIMD computations. We instantiate such frontend with commit-and-prove SNARKs as backends, thus obtaining more efficient commit-and-prove SNARK for all SIMD computations.

Let's again suppose the SIMD computation C consist of N identical copies of the sub-computation G. Let ℓ be our packing factor, we will split these N copies into N/ℓ batches, each batch consisting of ℓ copies. Below we describe the commit-and-prove SNARK for each batch.

Required Frontend/Backend Compilers:

- A *k-bounded* and *p-satisfiable* R1CS-type EQC compiler.
- The packing compiler CRT.Pack and fast ring emulator FRE.
- A commit-and-prove SNARK backend with respect to some vector commitment scheme (KeyGen, Commit) with native field $\mathbb{F}_{p^*}$.

Preprocessing Phase:

1. For $i \in [\ell]$, apply the EQC compiler to the program G with i^{th} input io^i, and let $\mathbb{X}_{G^i} = (A_G, B_G, C_G, \text{io}^i, m, n)$ be the resulting R1CS instance.
2. *Apply Packing Compiler:*
   Let $\mathbb{X}_C^q \leftarrow \mathsf{CRT.Pack}(\mathbb{X}_{G^1}, \ldots, \mathbb{X}_{G^\ell})$ be the packed instance.
3. FRE *Fitting Stage:*
   First assert that $k \cdot p^2 \cdot (q \cdot (m+n))^2 < p^*$. Then let $\mathbb{X}_{\mathsf{Fit}}^{p^*} \leftarrow \mathsf{FRE.Fit}(\mathbb{X}_C^q)$ be the fitted instance.

Commit-and-Prove Phase:

1. *Prepare Witness Values:*
   - Let $\mathbf{z}_i$ be the extended witness for $\mathbb{X}_{G^i}$. Prover packs these values as $\mathbf{z} = \mathsf{CRT.Pack}(\mathbf{z_1}, \ldots, \mathbf{z}_\ell)$.
   - Prover then computes a shift vector $\mathbf{k} \in \mathbb{Z}_q^m$ such that $(A'_G \cdot \mathbf{z}||\mathbf{k}) \circ (B'_G \cdot \mathbf{z}||\mathbf{k}) = (C'_G \cdot \mathbf{z}||\mathbf{k})$ holds over the integers. Now reset the extended witness as $\mathbf{z} = \mathbf{z}||\mathbf{k}$.
2. *Commit Phase:*
   Prover commits to extended witness as $c \coloneqq \mathsf{Commit}(\mathsf{ck}, \mathbf{z}||\mathbf{k})$.
3. FRE *Emulating Stage:*
   Both prover and verifier apply $\mathbb{X}_{\mathsf{Emulate}}^{p^*} \leftarrow \mathsf{FRE.Emulate}(\mathbb{X}_{\mathsf{Fit}}^{p^*})$.
4. *Prove Phase:*
   The prover uses the commit-and-prove SNARK to produce a proof π attesting that $\mathcal{R}_{\mathsf{R1CS}}(\mathbb{X}_{\mathsf{Emulate}}^{p^*}, \mathbf{z}||\mathbf{k}) = 1 \wedge c = \mathsf{Commit}(\mathsf{ck}, \mathbf{z}||\mathbf{k})$[3].

Fiat-Shamir Transform One caveat in the above protocol is that in the FRE emulating stage, the compiler needs to sample random vectors for Batch-PoSO. To make the prover and verifier agree on the randomness, we will instantiate the Fiat-Shamir transform by setting the randomness as the output of a cryptographic hash functions, which takes as input the previous commitment of prover's witness vector. The security can be proved in the random oracle model.

Security Proofs We defer the security proofs to appendix section 9.2.

Efficiency Gain Recall that R1CS frontend size is determined by the dimension of the matrices, and the number of non-zero entries in these matrices. For ease of illustration, here let's just consider the dimension. Without applying CRT.Pack and FRE, the naive R1CS frontend $\mathbb{X}_{G^1}, \ldots, \mathbb{X}_{G^\ell}$ has dimension $\ell \times (m \times n)$. Applying CRT.Pack and FRE yields a R1CS frontend $\mathbb{X}_{\mathsf{Emulate}}^{p^*}$ with dimension roughly $(m + O(\lambda)) \times (m+n)$. Therefore we have a dimension reduction of $O(\ell)$ as a result of applying those frontend compilers. We refer the reads to implementation section 8 for actual numbers obtained from concrete instances.

Succinct Verification The aforementioned construction is almost a commit-and-prove SNARK, except that the verification is not succinct. More specifically, this is due to the emulating stage of FRE: It involves sampling a random vector as long as the prover's witness size, then feeding it to a Batch-PoSO instance $\mathbb{X}_{\mathsf{PoSO}}$, and finally concatenating the instance with $\mathbb{X}_{\mathsf{Fit}}^{p^*}$. To deal with the this issue, we propose two modifications to FRE:

1. Instead of sampling a long random vector, we show how to reuse a short random vector while achieving the same statistical soundness error.

[3] In fact, the witness vector will be appended with additional witness involved in the range proof constraints in $\mathbb{X}_{\mathsf{Emulate}}^{p^*}$. The prover will commit to this additional witness as well and combine it with old commitment value c using the homomorphism in the vector commitment.

2. We notice that in almost all commit-and-prove SNARKs, the verifier only needs a vector commitment of any R1CS instance. Furthermore, this commitment scheme is additively homomorphic. This allows us to design a fast augmentation technique which enables quickly preparing the commitment instance $\mathbb{X}_{\mathsf{PoSO}}$, and concatenating it with commitment of $\mathbb{X}^{p^*}_{\mathsf{Fit}}$.

As a result of those two modifications, the verifier runtime only depends on λ, hence becoming succinct. We refer the readers to section 9.3 for more details.

Honest Verifier Zero-knowledge: If the underlying commit-and-prove SNARK satisfies honest verifier zero-knowledge, then so is our protocol.

7 Beyond SIMD: Highly Repetitive Computations

The aforementioned frontend compilers can be easily modified so as to support any highly repetitive computations. We again start with a simple example: Consider the incremental computation $C(x) = \underbrace{G(G \dots G(x))}_{\ell \text{ times}}$. Let's further denote by x_i the input to the i^{th} iteration of sub-computation G and y_i the corresponding output. That is, $G(x_i) = y_i$. Notice that due to the iterative structure, it is require that $x_i = y_{i-1}$ for all $i \in [\ell]$.

In the paradigm of EQC, let's add the following existentially quantified wire values $\{(x_i, y_i)\}_{i\in[\ell]}$. Then we can equivalently view the EQC of incremental computation as a SIMD-type EQC, plus additional consistency constraints:

- Consider the EQC for SIMD computation $C'(x_1, \dots, x_\ell) : (y_1 = G(x_1), \dots, y_\ell = G(x_\ell))$. For this component, we can use again use the packing compiler to pack all the $\ell \times$ R1CS instance into $\mathbb{X}^q_C \leftarrow \mathsf{CRT.Pack}(\mathbb{X}_{G^1}, \dots, \mathbb{X}_{G^\ell})$.
- Now let's add constraints to $\mathbb{X}^q_C$ which enforces consistency between transitions from output value y_i to input value x_{i+1}. However, since both values are packed, we must first unpack these values by adding the following:
  - **Unpacking Constraints:** $x = \sum_{i=1}^{\ell} x_i \cdot \lambda_i$, where λ_i's are apriori fixed Lagrange coefficients as defined in section 2.1. Similarly we add these unpacking constraints for y.
  - **Consistency Constraints:** $x_i = y_{i-1}$ $\forall i \in [\ell]$.

For any other highly repetitive computations, we can adopt similar methodology by adding these unpacking constraints and certain consistency constraints which enforce correct relationship between intermediate wire values.

8 Implementations and Evaluations

We implement our improved frontend compilers and evaluate its performance when instantiating with suitable commit-and-prove SNARK backends. Pursuing an application-level impact, we select SHA2-256 and SHA3-512 (keccak) instances used in Type-I zk-EVM projects for benchmarking our performance.

8.1 Experiment I: SHA2 and Keccak Speedup with Marlin

In this experiment we benchmark our frontend compiler as follows: We use SHA2-256 and SHA3-512 R1CS instances that are obtained from the `circomlib` package and github library. To further ensure that those instances are k-bounded, we make non-black box usage of `circom`'s code to modify those instances to ensure smallness of all matrix and witness entries. The experiment is taken over $N = (12, 48, 96, 192)$ of repeated copies of SHA2/SHA3 instances. For our method, we choose certain optimal packing factor ℓ, then apply our frontend compiler to output those N/ℓ packed instances as the frontend. For the *baseline method*, we consider the naive frontend compiler which directly outputs N instances in the library.

We then implement a modified version of Marlin's commit-and-prove protocol [CHM+20] in Rust from the Marlin backend arkworks library [ac22]. For the sake of estimating the real prover running time, we add the additional components of the protocol, including updating the commitment of R1CS instances in FRE.Emulate, on top of the Marlin prover. This is relatively lightweight, and the main prover cost comes from the additional non-zero entries and constraints that are added. To estimate the verifier runtime, we implement the verification procedure as base verification and a similar updation procedure for the R1CS commitment in FRE.Emulate. Importantly, this is a fixed cost regardless of the instance size. Finally, we benchmark the prover's runtime as well as its memory usage when proving circuits output by the two different methods above.

Hardware Setup We run this code on Amazon Linux EC2 instance (i4i.8xlarge) with 32 vCPUs and 256GB RAM. Note that we could run these programs on a machine with less memory, but this would incur swapping memory and thus affect benchmarking accuracy.

Determine Optimal Packing Factor In all our experiments, we use the elliptic curve `bls12-381` where the order of the scalar field is $p^* \approx 2^{254}$. Recall that in the emulating stage of FRE, in order for one single Batch-PoSO to achieve statistical soundness error of $\approx 1/p$, it is required that $kp^2(q(m+n)^2) < p^*$, where p arises from the *p-satisfiability* of R1CS compiler, k arises from its *k-bounded* property, (m, n) are the dimension of R1CS instance, and $q = \prod_{i=1}^{\ell} q_i$ is the product of primes used by the CRT packing compiler with $q_{\min} > p$. Thus to determine the optimal packing factor, one first needs to bound the size of q, and then rewrites q as the product of as many distinct primes as possible, so long as all primes are larger than p.

We notice that both SHA2-256 and SHA3-512 R1CS instances (with minor modifications) satisfy the *k-bounded* property with $k = 4$ and *p-satisfiability* with $p < 2^8$. Furthermore, both instances have dimensions satisfying $(m + n) < 2^{20}$. Using the condition that $kp^2(q(m + n)^2) < p^*$, one need to constrain $q < 2^{97}$. As a result, we can write q as a product of at most 12 primes that are larger than p. We set the statistical soundness error to be 2^{-70}.[4]

Evaluations We report the Marlin prover runtime as well as efficiency measurements of R1CS instances in Table 5.

	Marlin SHA2-256 **Packed/Baseline**			
Copies	P_{time}(s)	Mem Usage(GB)	Dim.	Non-Zero
12	96/204	3.46/4.8	70702/350785	886336/1618752
48	301/750	12.52/18	279076/1403137	3530461/6475008
96	543/1502	24.5/34.5	558151/2806273	7060911/12950016
192	1058/3010	47.15/69.3	1116301/5612545	14121811/25900032

	SHA3(Keccak) **Packed/Baseline**	
Copies	Dim.	Non-Zero
12	240238 / 1790976	5461867 / 8680200
24	479414 / 3600000*	10917131 / 17300000*
36	718590 / 5400000*	16402513 / 26000000*

	Nova IVC (N Steps of SHA2-256) **Packed/Baseline**			
Steps	Foldings	P_{time}(s)	Constraints	Variables
12	1/12	7.5/27.3	528766/6310452	1042870/6304344
24	2/24	13.8/51.3	1057532/12620904	2085740/12608688
48	4/48	26.1/99.6	2115064/25241808	4171480/25217376
96	8/96	51.2/196	4230128/50483616	8342960/50434752

Figure 5: Table values of prover time, memory usage, R1CS dimension (m, n), where m is number of constraints, and n is number of variables. "Dim" refers to $\max(m, n)$ and "Non-zero" refers total number of non-zero entries in A, B, C. *Estimated numbers.

[4]The computational soundness error is still 2^{-128}, due to underlying elliptic curve.

We obtain a final verification time of approximately 900 milliseconds, most of which is taken up by computing the updated commitment in FRE. Specifically this corresponds to an MSM of size 10000×11, which we fix to keep the verification time constant regardless of the instance size. This is a trade-off with the prover time, as decreasing the PoSO size helps verification, but increases the number of constraints and non-zero entries and affects the prover time.

The proof size of unmodified Marlin is 904 bytes, and our proof adds at most 4 additional group elements to the proof, bringing us to a larger but constant proof size around $1KB$.

8.2 Experiment II: Double Speedup with Nova

In this experiment we demonstrate that our improvement for repetitive computation is purely 'frontendish': More specifically, it does not overlap with any existing backend optimization techniques for repetitive computations. For this purpose, we select Nova[KST22], a heavily optimized backend targeting for incremental verifiable computation (IVC). We first give a brief overview of this scheme and show how to integrate our frontend compilers with Nova. We refer the readers to appendix section 9.4 for overview and corresponding implementation details.

The experiment is designed as follows: The goal is to prove an IVC consisting of $N = (12, 24, 48, 96)$ steps, where each step executes 20 copies of SHA2-256 instances. For our method, we split those N steps into $\ell \cdot (N/\ell)$ steps, where $\ell = 12$ is the same packing factor as in the previous experiment. Then we pack every ℓ steps using our frontend compiler, resulting in an IVC consisting of N/ℓ steps in total. For the *baseline method*, we consider the naive frontend compiler which directly outputs 20 copies per step, for all N steps. We also benchmark the prover time to notice the effect of our improvement on the final time. Furthermore, we benchmark the number of foldings required in Nova.

Hardware Setup We run this code on a machine with 16GB RAM and a 8-core AMD Ryzen 7 5800H CPU.

Evaluations For each IVC of N steps (proving N computations), we report the Nova prover runtime, the number of foldings required, as well as the total number of constraints and variables in the R1CS. The numbers are given in table 5.

References

[ac22] arkworks contributors. `arkworks` zksnark ecosystem, 2022.

[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`.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

[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`.

[BCL+21] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 681–710, Virtual Event, August 2021. Springer, Heidelberg.

[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.

[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.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS.

In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020.

[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.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In Shafi Goldwasser, editor, *ITCS 2012*, pages 90–112. ACM, January 2012.

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 769–793. Springer, Heidelberg, May 2020.

[FS01] Pierre-Alain Fouque and Jacques Stern. Fully distributed threshold RSA under standard assumptions. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 310–330. Springer, Heidelberg, December 2001.

[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`.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[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`.

[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`.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. `https://eprint.iacr.org/2019/953`.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *24th ACM STOC*, pages 723–732. ACM Press, May 1992.

[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.

[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.

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023. `https://eprint.iacr.org/2023/552`.

[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.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Shaun Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1074–1091, 2021.

[WYY+22] Chenkai Weng, Kang Yang, Zhaomin Yang, Xiang Xie, and Xiao Wang. AntMan: Interactive zero-knowledge proofs with sublinear communication. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2901–2914. ACM Press, November 2022.

[XZC+22] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkBridge: Trustless cross-chain bridges made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 3003–3017. ACM Press, November 2022.

[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.

[XZZ+19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 733–764. Springer, Heidelberg, August 2019.

[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.

9 Appendix

9.1 LCM Bound for Batch-PoSO

Lemma 9.1 (LCM bound for Batch-PoSO)**.** *Given* $\{z_i\}_{i\in[n]}$ *and* $z_{i,1} \in [-N, N], z_{i,2} \in [1, D]$ *such that* $z_i = z_{i,1} \cdot z_{i,2}^{-1} \mod p^*$*, define* $L \coloneqq LCM\{z_{i,2}\}_{i\in[n]}$*. Also suppose that* $L > D > 4$ *and that* $ND^n < p^*$*. Then, we additionally have*

$$\Pr_{r_1,r_2,\ldots,r_n\leftarrow[0,D)}\left[\sum_{i=1}^{n} r_i \cdot z_i < N \mod p^*\right] \leq \frac{1}{D} + \frac{b}{L}$$

where $b = \gcd(z_1, \ldots, z_n, L)$*.*

Importantly, when $b = 1$, this says that whenever the prover succeeds in the Batch-PoSO, then we must have $L < D$ with high probability.

Proof. Note that the condition in the probability statement above is equivalent to (for some $c \in [N]$)

$$\sum_{i=1}^{n} r_i \cdot z_i = c \mod p^* \implies \sum_{i=1}^{n} r_i \cdot z_{i,1}\frac{L}{z_{i,2}} = cL \mod p^*$$

where $L = LCM\{z_{i,2}\}_{i\in[n]}$. Here, both LHS and RHS are lesser than p^* (due to the condition in the statement of the lemma), hence this equation holds over $\mathbb{Z}$. We can rewrite this in the following form:

$$\sum_{i=1}^{n} r_i \frac{z_{i,1}}{z_{i,2}} \in \mathbb{Z} \implies \sum_{i=1}^{n} r_i z_{i,1}\frac{L}{z_{i,2}} = 0 \mod L$$

The proof is by induction. WLOG, let $b = 1$. If not, we can divide all coefficients and c by b, and consider the probability mod k/b. (If c is not divisible by b, the probability is 0, which is smaller than the required upper bound)

Consider the base case $n = 1$. Here,

$$\Pr_{r\leftarrow[0,D)}[rz = c \mod L] = \Pr_{r\leftarrow[0,D)}[r = cz^{-1} \mod L] \leq \frac{1}{D} + \frac{1}{L}$$

since z is invertible modulo L.

Suppose the statement is true for $n-1$. For n, notice that

$$\Pr_{r_1,r_2,\ldots,r_n\leftarrow[0,D)}\left[\sum_{i=1}^{n} r_i \cdot z_i = c \mod L\right] = \frac{1}{D}\sum_{\theta\leftarrow[0,D)} \Pr\left[\sum_{i=1}^{n-1} r_i \cdot z_i = c - z_n\theta \mod L\right]$$

Here, let $b' = \gcd(z_1, \ldots, z_{n-1}, L)$. Then, $\gcd(b', z_n) = b = 1$. Notice that any probability term in the above summation is zero if $c - z_n\theta$ is not divisible by b'. Since $c - z_n\theta$ is an arithmetic progression with common difference coprime to b', we can upper bound the number of θs for which $b' \mid (c - z_n\theta)$ - this is at most $\lceil D/b' \rceil$.

Hence, the above summation can be bounded as

$$\begin{aligned}
&\frac{1}{D}\sum_{\theta\leftarrow[0,D)}\Pr_{\mathbf{r}\leftarrow D^n}\left[\sum_{i=1}^{n-1} r_i\cdot z_i = c - z_n\theta \mod L\right] \\
&\leq \frac{1}{D}\cdot\left\lceil\frac{D}{b'}\right\rceil\cdot\left(\frac{1}{D}+\frac{b'}{L}\right) \\
&< \frac{1}{D}\cdot\left(\frac{D}{b'}+1\right)\cdot\left(\frac{1}{D}+\frac{b'}{L}\right) \\
&= \left(\frac{1}{b'}+\frac{1}{D}\right)\cdot\left(\frac{1}{D}+\frac{b'}{L}\right) \\
&= \frac{1}{L}+\frac{1}{D}\left(\frac{1}{b'}+\frac{b'}{L}+\frac{1}{D}\right)
\end{aligned}$$

Notice that when $L > D > 4$, we have

$$\frac{1}{b'}+\frac{b'}{L}+\frac{1}{D} < \frac{1}{2}+\frac{2}{L}+\frac{1}{D} < 1$$

This completes the proof. □

9.2 Security Proofs of SNARK for SIMD Computations

We show that the construction of SNARK for SIMD computations described in section 6 satisfies completeness and soundness.

Completeness Suppose that all ℓ copies of the sub-computation G is executed correctly. Due to the correctness of EQC compiler, we know that the $\ell \times \mathsf{R1CS}$ instances $\mathbb{X}_{G^1}, \ldots, \mathbb{X}_{G^\ell}$ are all satisfiable. Due to the correctness of CRT packing, we must have $\mathbb{X}_C^q$ being a satisfiable instance. Let $\mathbb{W}$ be its witness. It's easy to see that the prover can always compute some vector $\mathbf{k} \in \mathbb{Z}_q^m$ such that $(A\cdot\mathbf{z}||\mathbf{k}) \circ (B\cdot\mathbf{z}||\mathbf{k}) = (C\cdot\mathbf{z}||\mathbf{k})$ over the integers, where $\mathbf{z} = (1, \mathrm{io}, \mathbb{W}) \in \mathbb{Z}_q^{n+1}$. Since this equation holds over the integers, it also holds over modulo p^*. Thus the concatenated witness $\mathbb{W}||\mathbf{k}$ is a valid witness for the instance $\mathbb{X}_{\mathsf{Fit}}^{p^*}$. Furthermore, since $(\mathbb{W}||\mathbf{k}) \in \mathbb{Z}_q^{(m+n)}$ and $R = q$, all witness values are in the language $L_{R,1}$. Due to the completeness of Batch-PoSO, $\mathbb{X}_{\mathsf{PoSO}}$ must also be a satisfiable instance with respect to witness $\mathbb{W}||\mathbf{k}$. Therefore, the concatenated instance $\mathbb{X}_{\mathsf{Emulate}}^{p^*}$ consisting of $\mathbb{X}_{\mathsf{Fit}}^{p^*}$ and $\mathbb{X}_{\mathsf{PoSO}}$, must be satisfiable with respect to the same witness $\mathbb{W}||\mathbf{k}$. Therefore, the prover can always produce a valid SNARK proof π using this witness.

Soundness Assuming that $k\cdot q_{\min}^2\cdot(q\cdot(m+n))^2 < p^*$, we show that this construction has soundness error at most $1/2^\lambda$. Recall that the language $L_{N,D}$ is set to be $D = q_{\min}-1$, $N = q\cdot(q_{\min}-1)\cdot(m+n)$.

Let's first split into two cases:

- **Case I:** Assume the prover's concatenated witness $\mathbb{W}||\mathbf{k}$ do not belong to the language $L_{N,D}$. Then by soundness of Batch-PoSO, the prover can pass each Batch-PoSO protocol with probability at most $1/D$, thus the soundness error in this case is at most $\frac{1}{D}^{\lambda/\log(D)} = 1/2^\lambda$.
- **Case II:** Assume the prover's concatenated witness $\mathbb{W}||\mathbf{k}$ are all in language $L_{N,D}$. We argue that if the $\mathbb{X}_{\mathsf{Emulate}}^{p^*}$ is satisfiable, then all of $\mathbb{X}_{G^1}, \ldots, \mathbb{X}_{G^\ell}$ must also be satisfiable.

  Since $\mathbb{X}_{\mathsf{Emulate}}^{p^*}$ is satisfiable, this implies that $\mathbb{X}_{\mathsf{Fit}}^{p^*}$ must also be satisfiable. Consider each constraint $i \in [m]$ in $\mathbb{X}_{\mathsf{Fit}}^{p^*}$:

$$A'_G[i]\cdot\mathbf{z}\cdot B'_G[i]\cdot\mathbf{z} = C'_G[i]\cdot\mathbf{z} + \mathbf{k}[i]\cdot q \mod p^*$$

  Let $\mathbf{z}||\mathbf{k} \in \mathbb{Z}_q^{n+m}$ be the extended witness vector such that $\mathbf{z}$, $\mathbf{k}[i]$ will pass the above constraint. Since the extended witness is also in $L_{N,D}$, both $\mathbf{z}$ and $\mathbf{k}[i]$ admit representative in $\mathbb{Q}_{N,D}$. We

slightly abuse the notation and write $\frac{\mathbf{z_1}}{\mathbf{z_2}}$ as the corresponding vector of rational representatives for $\mathbf{z}$ (and similarly for $\mathbf{k}$), then we have:

$$A'_G[i] \cdot \frac{\mathbf{z_1}}{\mathbf{z_2}} \cdot B'_G[i] \cdot \frac{\mathbf{z_1}}{\mathbf{z_2}} = C'_G[i] \cdot \frac{\mathbf{z_1}}{\mathbf{z_2}} + \frac{\mathbf{k_1}[i]}{\mathbf{k_2}[i]} \cdot q \mod p^*$$

Let's multiply both sides by LCM 'L' of denominators, we have:

$$A'_G[i] \cdot \mathbf{z_1} \cdot B'_G[i] \cdot \mathbf{z_1} \cdot L = (C'_G[i] \cdot \mathbf{z_1} + \mathbf{k_1}[i] \cdot q) \cdot L \mod p^*$$

Let's first examine the left hand side. Since $\mathbb{X}^{p^*}_{\mathsf{Fit}}$ is also k-bounded, the value of $A'_G[i] \cdot \mathbf{z_1} \cdot B'_G[i] \cdot \mathbf{z_1}$ is at most k assuming $\mathbf{z_1} \in \mathbb{Z}_k^{n+1}$. In this case, we have the guarantee that $\mathbf{z_1} \in \mathbb{Z}_N^{n+1}$. Therefore this value will be blown up by at most a factor of $(N/k)^2$. Thus we can can still bound this value by $k \cdot (N/k)^2 < k \cdot D \cdot (q \cdot (m+n))^2$. Similarly, the right hand side value $C[i] \cdot \mathbf{z_1} + \mathbf{k_1}[i] \cdot q$ is at most $(k + q^2) \cdot D \cdot (m+n)$. We can always assume that left term $k \cdot D \cdot (q \cdot (m+n))^2$ dominates.

Furthermore, by the LCM lemma 9.1, L is at most D with all but negligible probability. As a result, both sides of equation have value at most $k \cdot D^2 \cdot (q \cdot (m+n))^2 < k \cdot q_{\min}^2 \cdot (q \cdot (m+n))^2 < p^*$. Therefore this equation must also hold over the integers. Let's divide back the LCM,

$$A'_G[i] \cdot \frac{\mathbf{z_1}}{\mathbf{z_2}} \cdot B'_G[i] \cdot \frac{\mathbf{z_1}}{\mathbf{z_2}} = C'_G[i] \cdot \frac{\mathbf{z_1}}{\mathbf{z_2}} + \frac{\mathbf{k_1}[i]}{\mathbf{k_2}[i]} \cdot q$$

Moreover, since $D = q_{\min} - 1$, by invertibility lemma 5.4, the following values are well defined: $\mathbf{z'} = \frac{\mathbf{z_1}}{\mathbf{z_2}} \mod q$. Now take the above equation over the ring $\mathbb{Z}_q$, we have:

$$A'_G[i] \cdot \mathbf{z'} \cdot B'_G[i] \cdot \mathbf{z'} = C'_G[i] \cdot \mathbf{z'} \mod q$$

Since this is true for eacg constraint $i \in [m]$, we have:

$$(A_G \cdot \mathbf{z'}) \circ (B_G \cdot \mathbf{z'}) = C_G \cdot \mathbf{z'} \mod q$$

Thus $\mathbb{X}^q_C = \mathsf{CRT.Pack}(\mathbb{X}_{G^1}, \ldots, \mathbb{X}_{G^\ell})$ is satisfiable. Therefore all of $\mathbb{X}_{G^1}, \ldots, \mathbb{X}_{G^\ell}$ are satisfiable.

9.3 Succinct Verification

The construction in section 6 does not meet succinct verification. More precisely, the verifier's running time in this protocol is linear in the length of the prover's witness. This is undesirable in many practical use cases where we demand succinct verification.

We observe that the 'non-succinctness' arises from the fact that the verifier needs to perform the following steps in the FRE emulating stage:

1. Let's denote by K the number of repetitions of Batch-PoSO. In this case $K = \lambda / \log(q_{\min})$. The verifier first needs to use hash function to generate K random vectors, each of which has length $(m+n)$.
2. The verifier needs to prepare the Batch-PoSO instance $\mathbb{X}_{\mathsf{PoSO}}$ and then concatenates it with $\mathbb{X}^{p^*}_{\mathsf{Fit}}$. The first instance has dimension $K \times (m+n)$ and the second has dimension $m \times (m+n)$.

For the sake of succinct verification, we want verifier's work to be independent of both m and n. In other words, its runtime should be $p(\lambda)$ for some apriori fixed polynomial $p(\cdot)$.

We now explain how to modify the FRE compiler in the emulating stage so as to meet succinct verification.

1. **Reusing Randomness:** In the Batch-PoSO protocol, the prover holds a witness vector $\mathbf{z}$ of length $m+n$. Instead of executing a single Batch-PoSO protocol on the vector $\mathbf{z}$, we will break $\mathbf{z}$ into $c = \frac{m+n}{p(\lambda)}$ number of chunks, each chunk with size $p(\lambda)$. Let $\mathbf{z} = \mathbf{z_1} || \ldots \mathbf{z_c}$. Then we execute a Batch-PoSO protocol for each chunk $\mathbf{z_i}$. Importantly, we sample only one random vector $\mathbf{r} \in \mathbb{Z}_D^{p(\lambda)}$, and reuse this vector across all c Batch-PoSO protocols. By union bound, this incurs a soundness error of at most c/D for the original vector $\mathbf{z}$. To amplify the final soundness error to $1/2^\lambda$, we will instead perform $K = \frac{\lambda}{\log(p) - \log(c)}$ repetitions of Batch-PoSO. To conclude, the randomness complexity can be reduced to $K \cdot p(\lambda) \approx O(p(\lambda))$.

2. **Fast Augmentation of R1CS Instance:** We suggest a fast augmentation technique to prepare, and then concatenate Batch-PoSO instance $\mathbb{X}_{\mathsf{PoSO}}$ with $\mathbb{X}_{\mathsf{Fit}}^{p^*}$. We observe that in almost all commit-and-prove SNARKs, the R1CS instances are given as vector commitments to the verifier, so as to achieve holography. For example, in [CHM+20] [COS20] [Set20], the commitment of R1CS is often done by first committing to the positions (indices) of all non-zero entries in the three (A, B, C) matrices, and then the values of these non-zero entries. As a result we can always assume that after the prepossessing phase, the $\mathbb{X}_{\mathsf{Fit}}^{p^*}$ is given to the verifier as a succinct commitment.

   Furthermore, most existing commit-and-prove SNARKs utilize a special type of vector commitment scheme that is linearly homomorphic. This suggests a fast augmentation method as follows:

   (a) During the preprocessing phase, we first commit to the positions of all non-zero entries of $\mathbb{X}_{\mathsf{PoSO}}$. Notice that this is doable since we can fix apriori the indices of non-zero entries of $\mathbb{X}_{\mathsf{PoSO}}$.

   (b) Recall that the non-zero entries of $\mathbb{X}_{\mathsf{PoSO}}$ (almost) entirely depend on the random vector $\mathbf{r}$. For each repetition $j \in [c \cdot K]$, the non-zero entries have the form $\mathbf{r^j}||\mathbf{r^j}||\ldots||\mathbf{r^j}$. Leveraging the homomorphic property of vector commitments, we can generate $p(\lambda)$ number of committed masks in the preprocessing phase, where these masks take the following form:

$$
\begin{aligned}
c_1 &= \mathsf{Commit}(\mathsf{ck}, [\underbrace{1,0,\ldots,0}_{p(\lambda)},\ldots\underbrace{1,0,\ldots,0}_{p(\lambda)}]),\\
c_2 &= \mathsf{Commit}(\mathsf{ck}, [\underbrace{0,1,\ldots,0}_{p(\lambda)},\ldots\underbrace{0,1,\ldots,0}_{p(\lambda)}]),\\
&\vdots\\
c_{p(\lambda)} &= \mathsf{Commit}(\mathsf{ck}, [\underbrace{0,0,\ldots,1}_{p(\lambda)},\ldots\underbrace{0,0,\ldots,1}_{p(\lambda)}]).
\end{aligned}
$$

   Now verifier can commit to the non-zero entries by computing $c_{\mathbf{r^j}} = \sum_{i=1}^{p(\lambda)} c_i \cdot \mathbf{r^j}[i]$. It thus takes only a total of $p(\lambda)$ operations to derive the commitment of a single repetition. In order to combine the commitments of non-zero entries for all repetitions of PoSO, we again use the homomorphic property of vector commitments. Overall, it takes the verifier $O(p(\lambda))$ operations to commit to all the non-zero entries in the instance $\mathbb{X}_{\mathsf{PoSO}}$. Now verifier is done with preparing the commitment of $\mathbb{X}_{\mathsf{PoSO}}$.

   (c) Finally, the verifier will concatenate the two committed instances $(\mathbb{X}_C^{p^*}, \mathbb{X}_{\mathsf{PoSO}})$ using the homomorphism of the vector commitment.

9.4 Integration with Nova

We choose a special backend named Nova [KST22], which is a folding scheme that targets for incremental verifiable computation (IVC). As pointed out in section 2.6, incremental computation is a common type of highly repeititive computation. Furthermore, IVC has been extensively used in many Type-I/II zk-EVM projects due to its ability to capture state transitions of RISC-V based machinery.

We choose Nova as our example in this section due to two reasons: Firstly, Nova (along with its following works [BC23, KS23]) maintains the state-of-art of proving incremental computation in terms of proving time. For this reason, its open source implementation has been deployed into certain zk-EVM projects. Therefore a fruitful combination of Nova with our improved frontend will likely make an impact towards further speeding up those zk-EVMs projects. Secondly, Nova also utilizes R1CS as its building block, which makes it conceptually simpler to combine Nova with our improved frontend. With that being said, we believe our frontend technique can be easily applied to constraint systems other than R1CS, such as CCS, introduced in [STW23], and thus be combined with other folding schemes [BC23, KS23].

9.4.1 A Brief Overview of Nova

Folding Scheme The core of Nova is a non-interactive folding scheme which folds two R1CS instances into one single instance. Importantly, the folded instance should encode the satisfiability of these two instances. Since the exact details of this scheme is irrelevant to our discussion, here we treat the non-interactive folding scheme as a black-box, and describe its grammar in the following figure 6:

Non-interactive Folding Scheme (NIFS)

- **Ingredients:**
  - A linearly homomorphic commitment scheme over message space $\mathbb{F}^n$. For any vector $\mathbf{m} \in \mathbb{F}^n$, we use $\overline{\mathbf{m}}$ to denote the commitment of such vector.
  - A hash function H (modeled as random oracle).
- **Inputs:**
  - Let there be two R1CS instances $\mathbb{X}_1 = (\mathbb{F}, A, B, C, \mathrm{io}_1 = x_1||y_1, m, n)$ and $\mathbb{X}_2 = (\mathbb{F}, A, B, C, \mathrm{io}_2 = x_2||y_2, m, n)$. Both $\mathbb{X}_1$ and $\mathbb{X}_1$ must have the same matrices A, B, C but potentially different inputs.
  - Let (w_1, w_2) be the corresponding witnesses and let $\mathbf{z_1} = (1, \mathrm{io}_1, w_1), \mathbf{z_2} \in \mathbb{F}^{n+1}$ be the corresponding extended witnesses. Let $(\overline{\mathbf{z_1}}, \overline{\mathbf{z_2}})$ be their commitments.
- **Outputs:**
  NIFS outputs the following:
  1. A folded instance $\mathbb{X}_{\mathbf{F}} = (\mathbb{F}, A, B, C, \mathrm{io}_{\mathbf{F}}, m, n)$, where the matrices (A, B, C) are unchanged.
  2. Folded (extended) witness and its commitment: $\mathbf{z_F}, \overline{\mathbf{z_F}}$. For sake of simplicity, we use the folded extended witness to represent the entire folded instance, since the matrices are always unchanged after each folding.

Figure 6: Simplified Description of Non-interactive Folding Scheme.

Remark 9.2. *For ease of explaining, we made a couple of simplifications in above descriptions. In fact, NIFS makes use of a "relaxed" type of R1CS which has a different formation of extended witness and additional error terms. However, we observe that these differences do not matter when combining with our frontends.*

9.4.2 From Folding Scheme to IVC

Assuming above NIFS, we can build an IVC with the following standard methodology. Let's suppose that the IVC corresponds to incremental computation $C(x) = \underbrace{G(G \dots G(x))}_{N \text{ times}}$.

Let's first apply the R1CS-type EQC compiler to G: for all $i \in [N]$, let $\mathbb{X}_{G_i} = (\mathbb{F}, A, B, C, \mathrm{io}_i, m, n)$ be the R1CS instance corresponding to the i^{th} invocation of G where the input/output is io_i. We set the initial input to be $\mathrm{io}_1 = x$ and final output to be $\mathrm{io}_N = C(x)$. For all other io_i $(1 < i < N)$, we shall let them unassigned for now. Intuitively, those intermediate input/output values will be assigned "on the fly" during consecutive foldings. Furthermore, notice that the (A, B, C) matrices are the same across all instances.

The next goal is to fold all the instances $\{\mathbb{X}_{G_i}\}_{i \in [N]}$ into one single instance using NIFS. The final instance shall encode the satisfiability of all N instances due to NIFS. As a result, we are only left with one instance in the end to prove satisfiability, whose size only depends on G, in particular being independent of the total number of copies $N \times |G|$, which fits the requirement of IVC.

In order to fold these instances, we define a new EQC G' which captures the "folding" aspect on top of G: First, it runs one more iteration of G, and second, folds one more R1CS instance with NIFS, where the instance is the result of applying the R1CS compiler to G' itself. To prevent the input size to G' from growing per folding, those inputs are always hashed per iteration. We give a simplified view of this EQC in figure 7.

From NIFS to IVC: The Augmented EQC G'

- **Preliminary:** Let there be some iterative computation $C(x) = \underbrace{G(G \dots G(x))}_{N \text{ times}}$.
- **Ingredients:**
  - The NIFS as described in figure 6.
  - Description of a hash function $H(\cdot)$.
- **Inputs:**
  - Some hash value h_i.
- **Non-deterministic Witness:**
  1. A round index number i indicating the i^{th} run of G.
  2. Some alleged input x_i corresponding to the i^{th} iteration input to G.
  3. A running folded instance $\mathbb{X}_{\mathbf{F}}$ which already folds all previous $i-1$ rounds G' instances, represented as $\mathbf{z_F}, \overline{\mathbf{z_F}}$.
  4. The current round G' instance, expressed as $\mathbf{z_i}, \overline{\mathbf{z_i}}$.
- **Program Specification:** Then the following steps are performed over the inputs and witnesses:
  1. Assert that $H(i, x_i, \mathbf{z_F}, \overline{\mathbf{z_F}}) = h_i$.
  2. Run one more iteration of G: $x_{i+1} \leftarrow G(x_i)$.
  3. Update the folding: $(\mathbf{z_F}, \overline{\mathbf{z_F}}) \leftarrow \mathbf{NIFS}((\mathbf{z_F}, \overline{\mathbf{z_F}}), (\mathbf{z_i}, \overline{\mathbf{z_i}}))$.
  4. Update the hash: $h_{i+1} \leftarrow H(i+1, x_{i+1}, \mathbf{z_F}, \overline{\mathbf{z_F}})$.
- **Outputs:**
  - The newest hash value h_{i+1}.

Figure 7: Simplified Description of Augmented EQC G'

9.4.3 Combining Improved Frontend with Nova

We now describe how to apply our improved frontend to Nova: First we show that our improved frontend is indeed "almost" compatible with NIFS, due to a nice property as we explain below. Next we show that with some little tweak in the EQC of G', we can make sure that our frontend is fully compatible with NIFS. Then one can follow the same approach to build a full-fledged IVC.

CRT Packing is Topology Preserving Our improved frontend consists of two components: $\mathsf{CRT.Pack}$ and FRE. We first observe that the CRT packing enjoys a useful property: It preserves the topology of circuits. More specifically, let's take $\ell \times \mathsf{R1CS}$ instances corresponding to the same computation G. Let (A, B, C) be the matrices in those R1CS instance (notice that all instances have same matrices). One may think of these matrices as capturing the topology of the circuit representation of G. Now observe that after applying $\mathsf{CRT.Pack}$ to these ℓ instances, the packed R1CS instance still admits the same matrices (A, B, C). This implies that the topology of circuit is preserved after packing.

Again, we will first split those N instances into N/ℓ batches and apply packing compiler and $\mathsf{FRE.Fit}$ to each batch of ℓ instances. This leaves us with $\mathbb{X}^1_{\mathsf{Fit}}, \dots, \mathbb{X}^{N/\ell}_{\mathsf{Fit}}$. Ideally, if we don't proceed the emulating stage of FRE, then we can just fold those packed N/ℓ instances via NIFS. Recall that one crucial requirement of NIFS is that the two folded instances must have the same topology (that is, they have the same (A, B, C) matrices). Critically, since CRT packing is topology preserving, our packed instances can still be folded via NIFS.

Dealing with $\mathsf{FRE.Emulate}$ The above approach almost works, however we have to also account for the effect of applying $\mathsf{FRE.Emulate}$. Unfortunately, naively applying this compiler breaks the topology-preserving guarantee: More specifically, for each packed instance $\mathbb{X}^i_{\mathsf{Fit}}$, $\mathsf{FRE.Emulate}$ requires sampling independent randomness $\mathbf{r_i}$. Then the $\mathsf{R1CS}$ matrices (A, B, C) of $\mathbb{X}^i_{\mathsf{Fit}}$ will be augmented with those randomness. Consequently, after this step, all these packed instances will have different matrices, thus no longer topology preserving.

To deal with this issue, we will reuse the same randomness across each batch of packed instance. This

will make sure that those packed instances still share the same set of matrices, thus compatible with NIFS.

Of course the prover must not learn the randomness before she sends over her commitment of the extended witnesses. Naively, she must send the commitments of all witnesses $\overline{\mathbf{z}_i}$ for all the $i \in [N/\ell]$ packed instances beforehand to the verifier. This would incur $O(N \cdot \lambda)$ communication, which becomes unacceptable for IVC. To solve this issue, we instead ask the prover to provide a hash chain of these commitments: $h \leftarrow H(H(H(\overline{\mathbf{z}_1}), \overline{\mathbf{z}_2}), \dots)$. Only the final hash value h is provided to the verifier, upon seeing which they will proceed with $\mathsf{FRE.Emulate}$ and sample one single piece of randomness that is reused across all packed instances.

Upon seeing the randomness, the prover will then update her extended witness for each packed instance. For all $i \in [N/\ell]$, let $\overline{\mathbf{z}_i}$ be the previous commitment of extended witness, and let $\overline{\mathbf{z}_i^*}$ be the updated commitment of extended witness.

Similar to the discussion in section 9.3, the prover must first show that $\overline{\mathbf{z}_i^*}$ and $\overline{\mathbf{z}_i}$ are consistent. In other words, $\overline{\mathbf{z}_i^*}$ is a commitment to the value $\mathbf{z}_i||\mathbf{v}||\mathbf{b}$. We refer to this part as consistency check. On top of this, prover must also prove the correct computation of hash chain.

Tweak G' with Hash Chain and Consistency Check To accommodate for above modifications, we need to slightly tweak the EQC G' to perform two additional checks at each step of folding:

- (Hash chain): Check that the hash is updated correctly: $h_{i+1} \leftarrow H(\overline{\mathbf{z}_i}, h_i)$.
- (Consistency check): Check that $\overline{\mathbf{z}_i^*}$ and $\overline{\mathbf{z}_i}$ are consistent.

The updated description of G' is provided in the following figure 8.

The Updated EQC G'

- **Inputs:**
  - Some hash value h_i.
- **Non-deterministic Witness:**
  1. A round index number i.
  2. Some alleged input x_i.
  3. A running folded instance $\mathbb{X}_{\mathbf{F}}$ represented as $\mathbf{z^*}_{\mathbf{F}}, \overline{\mathbf{z^*}_{\mathbf{F}}}$.
  4. The current round G' instance $\mathbf{z^*}_{\mathbf{i}}, \overline{\mathbf{z^*}_{\mathbf{i}}}$.
  5. The previous value in hashchain h_i^*.
  6. The commitment $\overline{\mathbf{z}_{\mathbf{i}}}$ which is used to form the hash chain.
  7. A proof π_i that $\overline{\mathbf{z}_{\mathbf{i}}}$ and $\overline{\mathbf{z}_{\mathbf{i}}^*}$ are consistent.
- **Program Specification:** Then the following steps are performed over the inputs and witnesses:
  1. Assert that $H(i, x_i, \mathbf{z}_{\mathbf{F}}, \overline{\mathbf{z}_{\mathbf{F}}}, h_i^*) = h_i$.
  2. Run one more iteration of G: $x_{i+1} \leftarrow G(x_i)$.
  3. Update the folding: $(\mathbf{z^*}_{\mathbf{F}}, \overline{\mathbf{z^*}_{\mathbf{F}}}) \leftarrow \mathbf{NIFS}((\mathbf{z^*}_{\mathbf{F}}, \overline{\mathbf{z^*}_{\mathbf{F}}}), (\mathbf{z^*}_{\mathbf{i}}, \overline{\mathbf{z^*}_{\mathbf{i}}}))$.
  4. Verify the consistency proof π_i.
  5. Include $\overline{\mathbf{z}_i}$ in the hash chain: $h_{i+1}^* \leftarrow H(\overline{\mathbf{z}_i}, h_i^*)$.
  6. Update the hash: $h_{i+1} \leftarrow H(i+1, x_{i+1}, \mathbf{z}_{\mathbf{F}}, \overline{\mathbf{z}_{\mathbf{F}}}, h_{i+1}^*)$.
- **Outputs:**
  - The newest hash value h_{i+1}.

Figure 8: Updated EQC G'

More Optimizations For the consistency proof π_i, one can rely on a KZG opening proof which makes the verification procedure into a pairing equation check. The pairing check, however, can introduce additional overhead in each folding. To remedy this overhead, we suggest use an accumulation scheme [BCL+21] to accumulate these pairing equations into one single pairing equation which will be checked by the verifier at the end.

Implementation Details We follow the aforementioned paradigm to modify Nova's folding scheme. More specifically, we first apply CRT.Pack and FRE to the R1CS instance and make sure to reuse the same randomness for each batch. Then for the sake of estimating the real prover runtime, we add the following constraints to the NIFS: 1. We append another hash chain using Poseidon hash function. 2. We add extra constraints to simulate the additional cost due to applying an accumulation scheme.