Improved SNARK Frontend for Highly Repetitive Computations

Sriram Sridhar[1] and Yinuo Zhang[1]

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

Abstract

Modern SNARK designs typically 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 satisfiability of the circuit. 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 representing a SHA-256 program as a circuit with pairing-based backend 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 SIMD computation with ≈ 180 SHA-256 instances, our improved frontend improves prover runtime by over $2.6\times$ and reduces memory usage by over $1.3\times$.

Central to our result and of independent interest, is an efficient technique for proving non-native modulo 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 attracted interests not just from theoretical standpoints (e.g. proving certain $\mathcal{NP}$ relations), but also 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 also maintaining sublinear verifier running time.

Bird's Eye View of Modern SNARK Paradigm: Frontend v.s. Backend The commonly used SNARKs are general-purpose and engineered to prove the correctness of (any) computer programs. At a high level, this is achieved by combining the following two steps:

First, the computer 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. Furthermore, it may consist of auxiliary input values which are called "non-deterministic advice", which helps keep the size of circuit relatively small and structurally simple. For example, when the computer program evaluates an inverse $x^{-1} \mod p$, the corresponding circuit representation can take the advice of the inverse (say y) and perform the check $x \cdot y = 1 \mod p$. This becomes a much cheaper task than to perform an inverse computation inside the circuit. More details on this are provided in Sec 3.6,3.6.1.

Subsequently, this circuit representation is fed into a SNARK to prove *circuit satisfiability.* This process is often referred to as the SNARK **backend**. Most aforementioned constructions of SNARKs are examples of such. Some popular ones [Gro16, CHM$^+$20, BCR$^+$19] target a special circuit representation called Rank-1 Constraint Systems (R1CS), while some others target circuit representations such as Plonk-style representation [GWC19, CBBZ22] or Algebraic Intermediate Representation (AIR) [BBHR19].

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 of prover efficiency 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 (see sec 3.5 for more details), 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 mainly measured through the prover's running time, relative to the size of the circuit representation. More specifically, for most popular SNARKs, the prover's work can be further divided into two categories: *Non-cryptographic operations:* These operations are relatively fast and inexpensive. Examples of such are computing non-deterministic advice in the circuit representation. *Cryptographic operations:* These operations are usually costly and expensive for various reasons. For example, in SNARKs which utilize a pairing-friendly cryptographic group [Gro16], those correspond to heavy operations such as multi-scalar exponentiations (MSM) in a group of large order. Other SNARKs [ZXZS20] require Merkle hashing and heavy FFT operations inside a large field. Regardless, the prover's running time in SNARKs is often dominated by these cryptographic operations which can be abstractly viewed as computing arithmetics over some very large field. More details on this can be found in remark 3.4.1. We point out that there exists certain SNARK constructions (also see sec 1.3 for such examples) where the backend prover does not need to operate over such large field. However, these SNARKs are not (widely) used in practice due to various reasons, hence not considered here.

Between Frontend and Backend: *The 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 arithmetics in some finite rings. The size of such rings are comparable to the size of the variables living inside the computer program. For example, the circuit representation of the SHA-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 prover is restricted to use much larger fields where these cryptographic operations need to be performed. For example, [Gro16] uses the field $\mathbb{F}_{p^*}$ where p^* is around 255 bit prime (i.e. $p^* \approx 2^{255}$), hence orders of magnitude larger than $\mathbb{F}_{253}$. To summarize, the backend supports proving circuit representations defined over a much larger field than those being supplied by the frontend.

This dichotomy is often solved by simply embedding the circuit representation (defined over some smaller field) inside the larger field. When the two fields substantially differ in size (as for most computer programs), this creates a noticeable inefficiency in SNARK backend. 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 consider this relation over certain gigantic field $\mathbb{F}_{p^*}$ (such as that being used by [Gro16] prover) 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 unused.

In this work we seek to incorporate this aspect into frontend design consideration. More specifically, consider a frontend which outputs some circuit representation for a certain program. Furthermore let this circuit be defined over some field $\mathbb{F}_p$. We wonder if one can devise an alternative circuit with smaller size by defining it over the larger backend field $\mathbb{F}_{p^*}$ where $p^* \gg p$. Since the backend prover operates in $\mathbb{F}_{p^*}$ and these operations dominate its running time, a smaller circuit would strictly improve its performance. This motivates us to study the following (informal) question:

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

1.1 Our Contributions

In this work we answer this question positively for a special class of programs which we call *highly repetitive* computations (see Sec 3.7 for more details). Informally speaking, a computation is highly repetitive if the same sub-computation is repeated multiple times with different inputs. One example of such programs is Same Instruction Multiple Data (SIMD) computation, which is ubiquitous in many real world applications. More concretely, our main contributions are as follows:

Techniques for Packing SIMD Circuits We introduce a packing technique for verifying SIMD computations. More specifically, consider any SIMD 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.[1]

Techniques for Efficiently Proving Non-native Modulo Arithmetic Crucial to our main results, and of independent interests, is an efficient embedding technique which allows the prover to embed elements of $\mathbb{Z}_q$ (where q is any large modulus) inside elements of $\mathbb{Z}_{p^*}$ (where p^* is any large prime such that $p^* \gg q$). Furthermore, due to this embedding, the prover can efficiently emulate $\mathbb{Z}_q$ arithmetic operations inside $\mathbb{Z}_{p^*}$.

Improved Frontend to R1CS for Highly Repetitive Computations To demonstrate our techniques, we propose a specific SNARK frontend which outputs an optimized R1CS instance for SIMD computations and more generally for highly repetitive computations. Compared with the R1CS instance outputted by most existing frontends, ours is constant times smaller yet defined a over much larger field. Our improvement may vary depending on the choice of backend fields and certain characteristics of repetitive computations.

Implementation of our Improved Frontend Our improved frontend can be combined with a class of popular backends which are called commit-and-prove SNARKs (see def 3.4). This direct combination yields succinct proofs but only linear verification. For the sake of sublinear verification we require certain additional properties of backend which are implicit in some existing schemes, Marlin [CHM+20] being one such example. We thus implement our improved frontend with Marlin, thereby deriving a full-fledged zk-SNARK.

1.2 Evaluations and Applications

Experiments We evaluate the performance of our improved frontend via the following experiments: we consider several instances of SIMD computation consisting of (11, 44, 88, 176) SHA-256 programs with 128-bit input.

For each experiment, we set the baseline as the following: we run a `circom` frontend to compile all n programs into a circuit, and then use the Marlin [CHM+20] backend to prove these circuits. We compare this with our improved frontend in the following way; we split the SHA-256 programs into (1,4,8,16) batches of 11 programs each (the number 11 is our packing factor throughout the paper, chosen to satisfy a constraint that depends on the compiler as well as the curve used; we use `bls12-381` for our experiments). For each batch, we use

[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.

Improvement factors				
n	P_{time}	Mem. Usage	Dim.	Non-zero
11	1.96	1.27	4.55	1.67
44	2.29	1.32	4.61	1.68
88	2.54	1.30	4.61	1.68
176	2.61	1.35	4.61	1.68

Figure 1: Improvement factors in prover time, memory usage, R1CS dimension and number of non-zero entries in the R1CS for our frontend compared to a naïve implementation for n repeated sub-computations of SHA-256 (with 128-bit input size).

`circom` [cir] to compile the computations in the batch into one circuit representation, and then use our improved frontend to pack all 11 circuits together into one single circuit. Then we run Marlin to prove the batched circuit.

We compare the performance of our improved frontend with the naïve frontend by comparing the prover running time and memory usage. We report the improvements in Table 1. More details can be found in Table 7 in Sec. 8.

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], 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 SHA-256). As another example, in many Proof-of-Stake blockchains, the proof involves verifying hundred of signatures. In Cosmos, each corresponds to an Ed-DSA signature, whose verification involves computing SHA-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). 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 replace the task of checking multiple bit-wise (binary) 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 improve the frontend design via the following way: 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: Firstly, the generation of look-up table is so expensive that in order to amortize this cost, the number of look-ups into the table must be comparable to its size. Hence this method is most effectively only for repeated binary operations with small size. In comparison, our method does not need a table and yields improvement even for small number of repetitions of large binary (or small arithmetic) computations. Secondly, look-up gates are specialized to the Plonk-style [GWC19] circuit representation (thus they do not support R1CS representation), whereas our method can be naturally extended to support all circuit representations.

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 (a type of sequentially repetitive computations) 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 tangential to our contributions. In fact we believe that our improved frontend (or at least our techniques) can be used in conjunction with those works to further speed up their prover performances.

2 Technical Overview

Improved SNARK Frontend From A Simple Example We give an overview of our improved frontend for highly repetitive computations based on a simple example. Consider the following SIMD computation C which computes the XOR of two length ℓ bit strings: $\mathbf{x} \in \{0,1\}^\ell,\ \mathbf{y} \in \{0,1\}^\ell :\ C(\mathbf{x},\mathbf{y}) \rightarrow \mathbf{z} = \mathbf{x} \oplus \mathbf{y} \in \{0,1\}^\ell$. We can view the computation C as ℓ parallel executions of the sub-computation G which outputs the XOR between two bits: $x_i \in \{0,1\},\ y_i \in \{0,1\} :\ G(x_i, y_i) \rightarrow z_i = x_i \oplus y_i \in \{0,1\}$. In order to express the correctness of computation C, we can simply express the correctness of each sub-computation G, as follows:

Let's consider an arithmetic circuit representation for the first sub-computation G_1 on input wires (x_1, y_1): It first checks all wires are 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:

- $x_1^2 - x_1 = 0$; (Which enforces x_1 to be binary.)
- $y_1^2 - y_1 = 0$; $z_1^2 - z_1 = 0$;
- $x_1 + y_1 - 2x_1 \cdot y_1 = z_1$. (Which enforces $z_1 = x_1 \oplus y_1$.)

Here we implicitly define these constraints over the ring of integers. Nonetheless, notice that it is indeed sufficient for them to hold over any prime field $\mathbb{F}_p$ such that $p \geq 2$, since the first three constraints implicitly ensure that all $x_1,\ y_1,\ z_1 \leq 1$ and $x_1 + y_1 - 2x_1 \cdot y_1 \leq 2 \leq p$. Therefore, if the fourth equation holds over $\mathbb{F}_p$, then it in fact holds over the integers.

On the other hand it is easy to see that $\mathbb{F}_2$ is the minimum field required to define this circuit representation. Let's thus rewrite its wiring constraints as follows:

- $x_1^2 - x_1 = 0 \mod 2$; $y_1^2 - y_1 = 0 \mod 2$;
- $z_1^2 - z_1 = 0 \mod 2$;
- $x_1 + y_1 - 2x_1 \cdot y_1 = z_1 \mod 2$.

In order to express the correctness of computation C, one can simply repeat the above constraints for each sub-computation G_i for $i \in [\ell]$. This simple approach yields a circuit representation of C consisting of 4ℓ constraints in total. Furthermore, the circuit can be defined over any field $\mathbb{F}_p$ for $p \geq 2$.

Nevertheless in most popular SNARKs, the backend prover must operate over some large field $\mathbb{F}_{p^*}$ (for example p^* can be some 255 bit prime). Our goal is to exploit this sheer size difference, and "pack" circuits of all sub-computation inside one single circuit over the larger field $\mathbb{F}_{p^*}$. To be more specific, we want to build another circuit for the same SIMD computation C which has *smaller size*, but instead requires a larger field to support.

The mathematical tool which we use for packing these circuits is Chinese Remainder Theorem (CRT), which states that for a set of ℓ coprime numbers $(q_1, \ldots, q_\ell)$, let q be their product (i.e. $q = \prod_{i=1}^{\ell} q_i$), then there exists the following ring isomorphism:

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

This suggests a natural way to pack all the ℓ circuit representations 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 let's consider the circuit representation of each sub-computation G_i defined over the quotient ring $\mathbb{Z}_{q_i}$ (which is the same as prime field $\mathbb{F}_{q_i}$). Since each $q_i \geq 2$, the corresponding circuit representation still expresses the correctness of the i^{th} sub-computation G_i.

Let q be the product of these ℓ primes. Instead of considering ℓ circuits with respect to arithmetic in ring $(\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_\ell})$, we use one single circuit with respect to the ring $\mathbb{Z}_q$:

- $x^2 - x = 0 \mod q$; $y^2 - y = 0 \mod q$;
- $z^2 - z = 0 \mod q$;
- $x + y - 2x \cdot y = z \mod q$.

Due to ring isomorphism, any wire values $(x, y, z \in \mathbb{Z}_q)$ satisfying the above circuit representation imply the following fact: for each $i \in [\ell]$, there exists corresponding wire values $(x_i = x \mod q_i,\ y_i = y \mod q_i,\ z_i = z \mod q_i)$ such that they satisfy the circuit representation (defined over $\mathbb{Z}_{q_i}$) for the sub-computation G_i. In other words, the aforementioned circuit is sufficient to express the correctness of SIMD computation C.

Let's recap what we have done so far, we start with a simple circuit representation consisting of ℓ identical sub-circuits of the same form and over the same field. Then we consider each circuit over a different prime field yet with the same guarantee of correctness, and finally use ring isomorphism to simulate all sub-circuits inside a larger field. During this process, we decrease the complexity of C by ℓ-fold compared to the simple representation we begin with.

Extend Our Frontend to Highly Repetitive Computations The above framework generalizes beyond this simple example of SIMD computation. In sec 6, we devise an improved SNARK frontend which can be applied to any SIMD computation. Our improvement is most significant when the sub-computation has relatively small variables, whereas the backend field has large size. One such example is proving multiple SHA-256 programs with pairing-based SNARKs. In sec 7.3 we extend this frontend to support the broader class of highly repetitive computations. Here we provide a quick overview. Let there be some SIMD computation C which consists of ℓ sub-computations: $\forall i \in [\ell] :\ G(x_i) \to y_i;$

First apply an existing frontend to the sub-computation G and retrieve the corresponding circuit representation $\overline{G}$. Let $\mathbb{F}_p$ be the minimum prime field which this circuit must be defined over (we call this notion p-satisfiability, see sec 3.6.1 for more details).

Then we pick ℓ different primes $\{q_i\}_{i\in[\ell]}$ such that each $q_i \geq p$. Our frontend compiler then packs (using the CRT) all sub-circuit-representations in the following way: The i^{th} sub-circuit $\overline{G}(x_i) \to y_i$ is lifted into ring $\mathbb{Z}_{q_i}$. Then all sub-circuits are packed into ring $\mathbb{Z}_q$. These steps happen only conceptually: No changes are required for the wiring structure of circuit $\overline{G}$. To facilitate this process, the prover and verifier will both pack all ℓ input and output wire values $\{(x_i, y_i \in \mathbb{Z}_{q_i})\}_{i\in[\ell]}$ into elements $x, y \in \mathbb{Z}_q$. Furthermore the prover will pack all the intermediate wire values as elements of $\mathbb{Z}_q$. Finally the prover proves the satisfiability of $\overline{G}$ with respect to input/output wires (x, y) using some existing SNARK backend.

There is yet a major problem associated with this approach: In most popular backends, the underlying field is chosen as certain fixed large prime number $p^* \neq q$ (since our q must be a composite). The native field only allows proving arithmetics over $\mathbb{Z}_{p^*}$, not over $\mathbb{Z}_q$. For large q (as in our case), it is not known how to efficiently prove these non-native arithmetic relations.

Proving Non-native Modulo Arithmetic As our second contribution, we propose an efficient "somewhat homomorphic" embedding scheme which allows the prover to embed an element of $\mathbb{Z}_q$ inside elements of $\mathbb{Z}_{p^*}$. Furthermore, due to the homomorphism, the prover can now efficiently emulate $\mathbb{Z}_q$ arithmetic operations inside $\mathbb{Z}_{p^*}$.

We present a high-level overview of our techniques here: Suppose we want the prover to prove an example constraint that $a \cdot b = c \mod q$. Instead of trying to prove this relation over modulo q, let's ask the prover to supply an additional shift value k and subsequently prove that $a \cdot b = c + k \cdot q$ (over $\mathbb{Z}$) , which amounts to a relation that holds over the integers. However, since the backend prover's operation is over $\mathbb{Z}_{p^*}$, this arithmetic relation must become $a \cdot b = c + k \cdot q \mod p^*$. Consequently, it does not necessarily imply that $a \cdot b = c \mod q$.

To circumvent this problem, we will in fact use a different representation of $\mathbb{Z}_{p^*}$ (and $\mathbb{Z}_q$) elements, which is called rational representatives (more details can be found in sec 4.2). Informally, we say that a rational number $\frac{a_1}{a_2}$ is a rational representative for an element $a \in \mathbb{Z}_{p^*}$ if it holds that $a = \frac{a_1}{a_2} \mod p^*$. For the sake of simplicity let's assume that both p^* and q are primes so that this relation is always well-defined. With such representation, the previous equation becomes:

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

where $(\frac{a_1}{a_2}, \dots, \frac{k_1}{k_2})$ are prover's witness values and q is the (fixed) modulus. The key observation is 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. If such products (both right and left hand sides) are less than p^*, then in fact the equation holds over the integers. Thus we can divide by the LCM, and get

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

as desired. The second trick now is to consider these rational numbers again as rational representatives, nonetheless of $\mathbb{Z}_q$ elements, instead of $\mathbb{Z}_{p^*}$ elements. This is equivalent to take this equation over modulo q. Concretely, let $a' = \frac{a_1}{a_2} \mod q$ (similarly for b, c, k), then it holds that:

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

which is the desired relation that we want to prove.

To summarize, whenever the $\mathbb{Z}_{p^*}$ elements (a, b, c, k) admit rational representatives such that the denominator and numerators are sufficiently small (we call such representatives bounded rational representatives), then in fact we can view these elements as embedded $\mathbb{Z}_q$ elements. Furthermore, in lemma 4.4 we show that such embedding is unique. Let S be the subset of $\mathbb{Z}_{p^*}$ where each element in S is some embedded $\mathbb{Z}_q$ element. Then the mapping induced by such embedding gives a natural (somewhat) homomorphism between the set S and $\mathbb{Z}_q$, thus allowing us to emulate $\mathbb{Z}_q$ arithmetics inside $\mathbb{Z}_{p^*}$.

Furthermore, we show that there exists a protocol (we call it Batch-PoSO) which allows the prover to efficiently prove that a set of elements belong to S. Moreover, this protocol can be incorporated into the SNARK frontend. More details of this protocol can be found in sec 4.3, 4.3.1.

More generally, for any $q << p^*$, where p^* is a fixed prime and q can be any large (composite) number, we show that there exists a set of bounded rational representative S' such that:

1. There exists a surjective mapping between S' and $\mathbb{Z}_q$.
2. There exists an injective mapping between S' and $\mathbb{Z}_{p^*}$.

Thus there exists a surjective mapping ϕ between a subset S^* of elements of $\mathbb{Z}_{p^*}$ and all elements of $\mathbb{Z}_q$. Furthermore, ϕ can be viewed as a (somewhat) surjective homomorphism (epimorphism) between the set S^* and $\mathbb{Z}_q$. Taking advantage of ϕ, the prover can emulate $\mathbb{Z}_q$ arithmetics inside $\mathbb{Z}_{p^*}$.

The main arithmetic intuition behind our improved frontend construction is as follows: The backend prover first proves that it only uses a subset S^* of $\mathbb{Z}_{p^*}$ elements which only contains valid embedding of $\mathbb{Z}_q$ elements. Moreover, this set membership is efficiently provable via certain appended frontend (Batch-PoSO). Then combining the somewhat epimorphic mapping $\phi : S^* \to \mathbb{Z}_q$, as well as the ring isomorphism $\mathbb{Z}_q \cong \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_\ell}$, the prover acquires access to a somewhat homomorphic mapping between $\mathbb{Z}_{p^*}$ and $(\mathbb{Z}_{q_1} \times \ldots \mathbb{Z}_{q_\ell})$, which allows her to emulate arithmetics in ℓ different finite rings using arithmetics in $\mathbb{Z}_{p^*}$.

Merging Our SNARK Frontend with Backends Our improved frontend can be applied to a wide range of SNARK constructions based on commit-and-prove protocols (see def 3.4). The combined protocol results in a commit-and-prove scheme with succinct proofs. Moreover it improves the prover running time and reduces its memory usage. Nevertheless, such base protocol incurs linear verifier running time. We show in sec 7.1 how to achieve succinct verification through simple modifications to the protocol which only makes non-black-box use of the vector commitment scheme and index relation of the underlying commit-and-prove protocol. Furthermore, we show how to turn our protocol into a full-fledged zk-SNARK using standard techniques in sec 7.2.

3 Preliminaries

Notation: We use λ for the security parameter. Let $\mathsf{negl}(\lambda)$ denote a negligible function. That is, for all polynomial $p(\lambda)$, it holds that $\mathsf{negl} < 1/p(\lambda)$ for large enough λ. We use $\mathbf{z}$ to denote a vector, $\mathbf{z}[i]$ to denote the i^{th} element in $\mathbf{z}$ and $\mathbf{z}[i:]$ to denote vector slicing from the index i. 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 $\mathbb{Z}_p$ denote the field $\mathbb{Z}/p\mathbb{Z}$, which corresponds to 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 a.

3.1 Chinese Remainder Theorem

Let $(q_1, \dots, q_n) \in \mathbb{Z}^n$ be a list of n coprime numbers. The Chinese Remainder Theorem (CRT) states that there exists the following ring homomorphism:

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

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

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

In particular, let $q = \prod_{i=1}^{n} q_i$, then it induces the following ring isomorphism:

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

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

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

where each coefficient λ_i here is an integer such that

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

We note that the coefficients λ_i could 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 $Q \cdot Q^{-1} = 1 \mod q_i$.

3.2 Vector Commitment Scheme

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

- $\mathsf{KeyGen}(1^\lambda)$: The commitment key generation algorithm take as input the security parameter, outputs a commitment key K and specifies an allowed message space $\mathbb{F}_{p^*}^n$.
- $\mathsf{Commit}(K, \mathbf{z}; r)$: The commitment algorithm takes as input a commitment key K, an vector $\mathbf{z} \in \mathbb{F}_{p^*}^n$, and a randomness r, and outputs a commitment c.

Furthermore, we require them to satisfy the following properties.

- **Succinct Commitments.** The size of commitment c is independent of the length of vector n.
- **Computational Binding.** For non-uniform probabilistic polynomial time algorithm A, there exists a negligible function $\nu(\lambda)$ such that

$$\Pr[K \leftarrow \mathsf{KeyGen}(1^\lambda), (\mathbf{z_1}, \mathbf{z_1}, r_1, r_2) \leftarrow A(1^\lambda, K) : \\ \mathbf{z_1} \neq \mathbf{z_2} \wedge \mathsf{Commit}(K, \mathbf{z_1}; r_1) = \mathsf{Commit}(K, \mathbf{z_2}; r_2)] \leq \mathsf{negl}(\lambda).$$

- **Optional Computational Hiding.** For any non-uniform probabilistic polynomial time distinguisher D, and any two vectors $\mathbf{z_1}, \mathbf{z_2}$, there exists a negligible function $\mathsf{negl}(\lambda)$ such that $|\Pr[K \leftarrow \mathsf{KeyGen}(1^\lambda) : D(1^\lambda, \mathsf{Commit}(K, \mathbf{z_1})) = 1] - \Pr[K \leftarrow \mathsf{KeyGen}(1^\lambda) : D(1^\lambda, \mathsf{Commit}(K, \mathbf{z_2})) = 1]| \leq \mathsf{negl}(\lambda)$. If the hiding property holds for any unbounded adversary, then we say the commitment scheme is *statistical* hiding.

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 us to add two commitments $\mathsf{Commit}(\mathbf{z_1}; r_1)$ and $\mathsf{Commit}(\mathbf{z_2}; r_2)$ to obtain a new commitment $\mathsf{Commit}(\mathbf{z_1} + \mathbf{z_2}; r_1 + r_2)$.

3.3 Interactive Proof Systems

Interactive Proofs. An interactive proof system for a language $L \in \mathcal{NP}$ in the CRS model is a pair of algorithms $(\mathsf{Setup}, \mathcal{P}, \mathcal{V})$, where the setup algorithm takes as input a security parameter λ, and outputs a crs, $\mathcal{P}$ takes as input the crs, an instance $\mathbb{X} \in L$, and a witness w for $\mathbb{X}$. The verifier takes $\mathbb{X}$ as input. The prover tries to convince the verifier that $\mathbb{X} \in L$, by interacting with it in multiple rounds. Furthermore, we require $\mathcal{P}, \mathcal{V}$ to satisfy the following properties.

- **Completeness.** For any instance $\mathbb{X} \in L$, and any witness w of x, we have $\Pr[\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda), \mathcal{P}(\mathsf{crs}, x, w) \leftrightarrow \mathcal{V}(\mathsf{crs}, x) : \mathcal{V} \text{ accepts}] = 1$.
- **Soundness.** For any non-uniform PPT malicous prover $\mathcal{P}^*$, there exists an negligible function $\mathsf{negl}(\lambda)$ such that

$$\Pr[\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda), x \leftarrow \mathcal{P}^*(\mathsf{crs}), \\ \mathcal{P}^*(x) \leftrightarrow \mathcal{V}(\mathsf{crs}, x) : x \notin L \wedge \mathcal{V} \text{ accept}] \leq \mathsf{negl}(\lambda).$$

Public-coin. A public-coin interactive protocol $(\mathcal{P}, \mathcal{V})$ is an interactive protocol where the verifier's random coins are public to the prover. Namely, each of verifier's message is a uniform random string and the verification process is public.

Argument of Knowledge. A public-coin interactive argument of knowledge for a language $L \in \mathcal{NP}$ is a pair of algorithms $(\mathsf{Setup}, \mathcal{P}, \mathcal{V})$,

- **Perfect Completeness.** For any adversary A, we have

$$\Pr[\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda), (x, w) \leftarrow A(1^\lambda, \lambda), \\ \mathcal{P}(\mathsf{crs}, x, w) \leftrightarrow \mathcal{V}(\mathsf{crs}, x) : \mathcal{V} \text{ accept } \vee (x, w) \notin \mathcal{R}_L] = 1,$$

  where $\mathcal{R}_L$ is the NP-relation of L.
- **Knowledge Soundness.** For all non-uniform PPT malicious prover $\mathcal{P}^*$, there exists an expected polynomial time extractor $\mathcal{E}$ such that for non-uniform PPT adversary A, there exists a negligible function $\mathsf{negl}(\lambda)$ such

that

$$\left| \Pr\left[\begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda), (\mathbb{X}^*, r^*) \leftarrow A(1^\lambda, \mathsf{crs}), \\ \langle \mathcal{P}^*(\mathsf{crs}, \mathbb{X}^*; r^*), \mathcal{V}(\mathsf{crs}, \mathbb{X}^*) \rangle \end{array} : \mathcal{V} \text{ accept} \right] - \right.$$
$$\left. \Pr\left[\begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda), (\mathbb{X}^*, r^*) \leftarrow A(1^\lambda, \mathsf{crs}), \\ (\mathbb{W}) \leftarrow \mathcal{E}^{\mathcal{O}}(\mathsf{crs}, \mathbb{X}^*) \end{array} : \begin{array}{c} \mathcal{V} \text{ accept } \wedge \\ (\mathbb{X}^*, \mathbb{W}) \in R \end{array} \right] \right|$$
$$\leq \mathsf{negl}(\lambda),$$

where r^* is the randomness of $\mathcal{P}^*$, and

$$\mathcal{O} = \langle \mathcal{P}^*(\mathsf{crs}, \mathbb{X}^*; r^*) \leftrightarrow \mathcal{V}(\mathsf{crs}, \mathbb{X}^*) \rangle$$

is the execution between the malicious prover $\mathcal{P}^*$ and the verifier $\mathcal{V}$. The adversary A is allowed to rewind the prover $\mathcal{P}^*$ to some point and resume it with fresh randomness from this point onwards.

Honest Verifier Zero-Knowledge. We say an public-coin interactive protocol in CRS model $(\mathsf{Setup}, \mathcal{P}, \mathcal{V})$ for a language L is honest-verifier zero-knowledge, if there exists a PPT simulator S such that, for any instance $\mathbb{X} \in L$, and any witness w for $\mathbb{X}$,

$$\{\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda), t \leftarrow \langle \mathcal{P}(\mathsf{crs}, \mathbb{X}, w), \mathcal{V}(\mathsf{crs}, \mathbb{X}) \rangle : (\mathsf{crs}, t)\}_{\lambda, \mathbb{X}}$$
$$\approx \{S(1^\lambda, \mathbb{X})\}_{\lambda, \mathbb{X}},$$

where t is the transcript of the protocol.

3.4 Commit-and-Prove Protocols

A public-coin commit-and-prove protocol for a vector commitment scheme $(\mathsf{Setup}, \mathsf{Commit})$ and a language $L \in \mathcal{NP}$ is a public-coin interactive proof system $(\mathsf{Setup}, \mathcal{P}, \mathcal{V})$ for the following language,

$$\mathcal{L}_K(L) = \{(c, \mathbb{X}) \mid \exists (w, r) : \mathcal{R}_L(\mathbb{X}, w) = 1 \wedge c = \mathsf{Commit}(K, w; r)\},$$

where $\mathcal{R}_L$ is the NP-relation for L, and $K \leftarrow \mathsf{Setup}(1^\lambda)$ is generated by KeyGen.

Commit-and-Prove SNARK A commit-and-prove SNARK is a succinct non-interactive argument of knowledge (SNARK) if it is further equipped with the following properties:

- **Non-Interactive:** The prover only sends a single proof (denoted by π) to the verifier and no further interaction is required.
- **Succinctness:** We say that the proof is succinct if the length of π is independent of the size of the instance $|\mathbb{X}, w|$. Furthermore, we say that the verification is succinct if the verifier's runtime is also sublinear in the size of the instance.
- **Argument of Knowledge.** The commit-and-prove protocol must be an argument of knowledge.

3.4.1 Field choice

Most existing commit-and-prove SNARKs are designed to support proving languages involving arithmetic relations over certain field. Nonetheless, depending on the construction of underlying vector commitment schemes, the choices of such field can vary.

Since the field choice plays an important role in this work, here we give a brief review of different vector commitment schemes and the corresponding field choices. Readers may also consult the survey of [Tha23], Ch.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 among the most popular ones and they are widely adopted in many real-world applications. However, the choice of field is very limited due to many required properties of the group. In these popular curves such as BLS12-381 and BN-254, the group order p^* is a fixed $\approx$ **255**-bit prime number. This also constraints the field choice to be $\mathbb{F}_{p^*}$.
- 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 is rather flexible but often needs to support FFT operations and p^* is often chosen as a fixed $\approx$ **64**-bit prime number.
- The third category utilizes hardness of unknown order groups [BFS20, CFKS22, AGL$^+$23, SB23]. These systems are not widely used in practice due to slow running time hence not of our interests.

In our work we mainly consider the first category as they capture a wide range of SNARKs deployed in practice. In the scope of this work, we only point out that since the prime p^* is very large, each group operation becomes very slow. For this reason the prover's efficiency in these systems is often dominated by the number of group operations that she needs to perform.

3.5 Rank-1 Constraint Systems

Definition 3.1 (R1CS)**.** *Rank-1 Constraint Systems is a type of commonly used arithmetic EQC in cryptographic proof systems. An* R1CS *instance is a tuple* $\mathbb{X} = (\mathbb{F}, A, B, C, io, m, n)$ *where* io *denotes the public input and output of the instance, and* $A, B, C \in \mathbb{F}^{m\times(1+n)}$ *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* $z = (1, io, w) \in \mathbb{F}^{n+1}$, $\cdot$ *is the matrix-vector product and* $\circ$ *is the Hadamard (entry-wise) product. We denote by* $\mathcal{R}_{\mathsf{R1CS}}(\mathbb{X}, w) = 1$ *in this case.*

Dear to our interest is another property: 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 in* $[0, k]$ *(where we must consider the operation* $(A \cdot z) \circ (B \cdot z) = (C \cdot z)$ *over the integers).*

3.6 Existentially Quantified Circuits

In this work we consider existentially quantified circuits (EQCs) introduced in [OBW22]. EQCs are circuits which consist of sets of wires taking values from some domain (such as bits in boolean circuits) and constraints that express certain relationships among wire values (such as the constraint $x \wedge y = z$). 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.

EQCs capture non-deterministic, non-uniform computation. Their non-determinism stems from the existentially quantified inputs whose values are, in principle, "guessed" by the execution substrate. Their non-uniformity reflects the fact that a circuit of a given size encodes a computation for a fixed-size input.

We especially consider the family of arithmetic EQCs where the constraints are over the arithmetic operators $(*/\cdot, +, -)$ corresponding to multiplication/addition/subtraction.

In particular, boolean circuits are such examples where for each of AND, OR, XOR and NOT gates, we can consider these analytical representations:

- $x \wedge y = xy$
- $x \vee y = x + y - xy$
- $x \oplus y = x + y - 2xy$
- $\neg x = 1 - x$

3.6.1 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 original computer program. In this work we specifically consider the EQC compiler that outputs an $\mathsf{R1CS}$ instance:

Definition 3.2 ($\mathsf{R1CS}$-type EQC compiler)**.** *Let P be a computer program which takes some value x as input, and let y be the alleged output. A $\mathsf{R1CS}$-type EQC compiler takes as input the program P, its input/output values (x, y) and produces an $\mathsf{R1CS}$ 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 define the following two additional properties of EQC compiler which meet our interests:

- **k-bounded**: The compiler always output a *k-bounded* $\mathsf{R1CS}$ instance.
- **p-satisfiable**: The compiler output a $\mathsf{R1CS}$ instance over certain field $\mathbb{F}_p$ (i.e. $\mathbb{X} = (\mathbb{F}_p, A, B, C, \dots)$). Furthermore, for every $p' \geq p$, 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, MD5 and SHA-256 computations, these EQC compilers are naturally k-bounded and p-satisfiable for small values of k and p (for example, in AES and SHA-256, [cir] admits $k = p = 2^8$). For all other programs, these compilers can be made to satisfy these two properties (still with small values of k and p) by allowing to output larger EQC instances.

Remark 3.3 (**Backend** v.s. **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*** *(such as the role of aforementioned EQC compiler). These instances are then consumed by various SNARKs schemes targeting for circuit satisfiability. This process of proving EQC instance is often referred to as the SNARK* ***backend****.*

3.7 Highly Repetitive Computation

Highly repetitive computation refers to computations where the same sub-computation is applied to multiple pieces of input which may or may not depend on each other. For example, in recursive programs certain sub-computation might be repeated for many times, each time taking the previous output as the next input.

For example, consider the incremental computation C where the sub-computation G is repeated ℓ times iteratively: That is: $C(x) = \underbrace{G(G \dots G(x))}_{\ell \text{ times}}$. SIMD computation can be viewed as a special case of highly repetitive computation.

3.7.1 Data Parallel Computation

Data parallel computation, or same instruction multiple data (SIMD) refers to computations where the same sub-computation is applied independently to multiple pieces of input data. This format of computation is ubiquitous in many real world applications. In this work we often denote the sub-computation by G, which could be either some computer programs or some arithmetic circuits.

For 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: $C(x_1, \dots, x_\ell) = (G(x_1), \dots, G(x_\ell))$.

4 Building Blocks

4.1 CRT Packing

Definition 4.1 (CRT Packing Scheme)**.** *Let there be a set of coprime numbers $(q_1, \dots, q_n)$ 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:*

- $\mathsf{CRT.Pack}(a_1, \dots, a_n) \to a$*: The packing algorithm takes as input $a_i \in \mathbb{Z}_{q_i}$ and outputs a number $a \in \mathbb{Z}_p$.*

- $\mathsf{CRT.Unpack}(a) \to (a_1, \ldots, a_n)$: *The unpacking algorithm takes as input some number $a \in \mathbb{Z}_q$ and outputs a set of n numbers $(a_1, \ldots, a_n)$ where $a_i \in \mathbb{Z}_{q_i}$ for every $i \in [n]$.*

As described in Sec. 3.1, the packing and unpacking algorithm are naturally defined by the ring isomorphism $\mathbb{Z}_q \cong \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_n}$.

4.2 Rational Representative

Consider the quotient ring $\mathbb{Z}_p$ for any prime number $p \in \mathbb{Z}$. While it is natural to represent every element in $\mathbb{Z}_p$ as integers in $\mathbb{Z}$ ranging from $[0, p-1)$, in [CKLR21] the authors propose an alternative representative, which they call "rational representative". The goal of using this representative is to simplify the task of proving non-native arithmetic operations in cryptographic proofs (jumping ahead this becomes one of the core problems in this work). In the original context of [CKLR21] this turns out to be very useful in the setting of performing range proofs.

Following the notations of [CKLR21, CGKR22], we provide the following definition of rational representatives:

Definition 4.2 (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 $x \in \mathbb{Z}_p$, 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 $x \in \mathbb{Z}_p$, it can have multiple rational representatives. Nevertheless in many cryptographic protocols, it is often desired that this representative be unique (i.e. any $x \in \mathbb{Z}_p$ cannot be represented by more than one rational). Therefore, the authors in [CKLR21] consider a smaller subset of rationals: $S \subseteq \mathbb{Q}$. In this subset S, each $x \in \mathbb{Z}_p$ has at most one (unique) representative in the set.

Such uniqueness does not come for free, it comes at the cost that certain elements $x' \in \mathbb{Z}_p$ do not admit any representative in the set S. Nevertheless, with some careful construction of such set, the elements of our interest will always admit representatives. One example is the set of Bounded Rational Representative:

Definition 4.3 (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:*

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

In this case the elements of our interest are all number in $[0, N] \subset \mathbb{Z}_p$. The cardinality of such set depends on both N and D, and when both are small enough, it can be shown that each element in $\mathbb{Z}_p$ admits unique representative.

Lemma 4.4 (Critierion for unique representative in $\mathbb{Q}_{N,D}$). *Let N, D be so that $N \cdot D < p/2$. Then for any $x \in \mathbb{Z}_p$, if there is a representative in $\mathbb{Q}_{N,D}$ of x (i.e. there exists some $\frac{n}{d}$ so that $x = n \cdot d^{-1} \mod p$), then $\frac{n}{d}$ must be unique.*

Proof. Suppose for the sake of contradiction that there exists two rationals $\frac{n_1}{d_1}, \frac{n_2}{d_2}$ such that $\frac{n_1}{d_1} \neq \frac{n_2}{d_2}$, and furthermore $x = \frac{n_1}{d_1} = \frac{n_2}{d_2} \mod p$. Multiplying by the common denominator $d_1 \cdot d_2$, we have $n_1 \cdot d_2 = n_2 \cdot d_1 \mod p$. However, since $N \cdot D < p/2$, it in fact holds that $n_1 \cdot d_2 = n_2 \cdot d_1$ over $\mathbb{Z}$ (not just modulo p). Thus $\frac{n_1}{d_1} = \frac{n_2}{d_2}$, which is a contradiction. □

4.3 Proof of Short Opening (PoSO)

In [CKLR21], the authors also designed a companion protocol which allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ (in zero-knowledge) that a committed value $x \in \mathbb{Z}_p$ admits a rational representative in $\mathbb{Q}_{N,D}$ for any choice of (N, D). Such a protocol is called a Proof of Short Opening (PoSO). The name arises from the fact that the prover shows the existence of some 'short' (bounded) rational representative that can be used to open the commitment (hence short opening).

The PoSO protocol is one of our main building blocks, so we will explain it in more details in this section. At a high level, PoSO mimics a Sigma protocol for proving knowledge of a committed value. Here we implicitly assume that the allowed message space in the commitment scheme is $\mathbb{Z}_p$. In order to show that a commitment $c = \mathsf{Commit}(x)$ (where $x \in \mathbb{Z}_p$) admits short opening, the prover samples a random mask $\tilde{x} \in [L]$ (where L is just large enough to hide the value x) and sends its commitment $\tilde{c} = \mathsf{Commit}(\tilde{x})$ to the verifier. The verifier then responds with a small random challenge $\gamma \in [D]$, and the prover then sends the value $v := x \cdot \gamma + \tilde{x}$ (as well we the underlying commitment randomness) to the verifier. The verifier then checks that v is a valid opening for the combined commitment $\tilde{c} + c$ (assuming the commitment scheme is additive homomorphic, see Sec 3.2), and further checks that $v \in [N]$.

To see that this protocol indeed proves short opening of x, observe that via forking lemma [PS01], one can extract the committed value x as $x = \frac{v_1 - v_2}{\gamma_1 - \gamma_2} \mod p$, where the numerator is in the range $[-N, N]$ and the denominator is in the range $[-D, D]$.

4.3.1 Batch Proof of Short Opening (Batch-PoSO)

While in [CKLR21] the role of PoSO is to prove in zero-knowledge the shortness of a single element, it does not fit our primary interest in this work. Instead we seek for a batched variant of PoSO which allows a prover to convince the verifier that, for some vector $\mathbf{z} \in \mathbb{Z}_{p^*}^n$ that is committed as $c = \mathsf{Commit}(\mathbf{z})$ via a vector commitment, each entry $\mathbf{z}[i]$ in the vector admits short opening. In particular, we require that the proof must be succinct (we do not require zero-knowledge). That is, the proof size is independent of the dimension of the vector. We call such protocol Batch-PoSO. We provide its formal definition as follows:

Definition 4.5 (Batch-PoSO)**.** *A Batch Proof of Short Opening (Batch-PoSO) is a commit-and-prove protocol for the following promise 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 for all $i \in [n]$, the entry $\mathbf{z}[i] \in [0, R]$.*
- *A vector $\mathbf{z} \in \mathbb{Z}_p^n$ is said to be in the language $L_{N,D}$ if for all $i \in [n]$, the entry $\mathbf{z}[i]$ admits a rational representative in $\mathbb{Q}_{N,D}$.*

Remark 4.6. *We remark that by using promise language for Batch-*PoSO*, we allow a gap between completeness and soundness: For completeness, we only require honest prover being able to prove that each element* $\mathbf{z}[i]$ *in the vector is in the range* $[0, R]$*. Whereas for soundness, any prover cannot succeed whenever the opening has numerator beyond the range* $[-N, N]$ *or denominator beyond range* $[-D, D]$ *(a.k.a. does not admit bounded rational representative).*

A protocol for Batch-PoSO is given in [CGKR22] and implicitly given in [GJJZ22]. We first present a high level sketch of the protocol. The core idea is to mimic the basic PoSO protocol (described above). The prover first sends the verifier the commitment $c = \mathsf{Commit}(\mathbf{z})$ which she aims to prove short opening. Now the main difference from PoSO is that instead of asking the verifier to send the challenge as a single short element, the 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. $\forall i \in [n], \mathbf{r}[i] \in [0, D)$). Furthermore, no mask is required from the prover's side since we don't require zero-knowledge. Upon receiving the random vector, the prover then computes the inner product $v :=< \mathbf{z} \cdot \mathbf{r} >$ over $\mathbb{Z}_{p^*}$ and sends v along with a proof π of the correctness of this inner product to the verifier (This can be achieved by combining with many existing commit-and-prove schemes which has succinct proofs). The verifier then accepts if both that π is a valid proof and that $v \in [N]$.

The formal description of Batch-PoSO is provided in Fig 2.

Batch-PoSO

- **Ingredients:** A commit-and-prove protocol (with succinct proofs) for proving arithmetic relations over $\mathbb{Z}_{p^*}$, where p^* is a fixed prime determined by the underlying vector commitment scheme.
- **Preliminary:** The prover holds some vector $\mathbf{z} \in \mathbb{Z}_{p^*}^n$. Both the prover and verifier receive (R, N, D) which defines the promise language $(L_{R,1}, L_{N,D})$.
- **Construction:**
  1. The prover generates $c := \mathsf{Commit}(\mathbf{z}; u)$ with some randomness u, and sends 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, who computes $v :=< \mathbf{z} \cdot \mathbf{r} >$
  3. The prover and verifier prepares the following R1CS instance (for proving correctness of above inner product): $\mathbb{X}_{\mathsf{PoSO}} = (A, B, C, \text{io} = [1, v], 1, n+1)$, where $A = [\mathbf{r}[1], \ldots, \mathbf{r}[n], 0, 0]$, $B = [0, \ldots, 0, 1]$ and $C = [0, \ldots, 1, 0]$.
  4. The prover and verifier execute the commit-and-prove protocol with respect to the R1CS instance $\mathbb{X}_{\mathsf{PoSO}}$ and commitment c.

$$\mathcal{P}\left(1^\lambda, (\mathbb{X}_{\mathsf{PoSO}}, u)\right) \leftrightarrow \mathcal{V}(1^\lambda, \mathbb{X}_{\mathsf{PoSO}}).$$

  If the prover succeeds in the commit-and-prove protocol and that $v \in [N]$, the verifier accepts, otherwise it rejects.

Figure 2: Description of Batch-PoSO.

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

Claim 4.7 (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 $v := < \mathbf{z} \cdot \mathbf{r} > \leq R \cdot D \cdot n \leq N$. Thus $v \in [N]$ and the verifier will always accept (assuming the commit-and-prove protocol also has perfect completeness). □

Claim 4.8 (Soundness of Batch-PoSO)**.** *For any fixed constant N, and for any $\{z_i\}_{i \in [n]}$ with $z_i \in \mathbb{Z}_p$ 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 $z_{i,1} \in [-N, N], z_{i,2} \in [1, D]$ such that $z_i = z_{i,1} \cdot z_{i,2}^{-1} \mod p^*$. Thus the soundness error of Batch-PoSO is at most $1/D$ (plus the soundness error of the underlying commit-and-prove protocol of inner product relation).

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

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

Let $z_{i,2} := r_i^1 - r_i^2 \in [1, D)$, thereby, let

$$z_{i,1} := \left(\sum_{j \neq i}^{n} r_j^* \cdot z_j + r_i^1 \cdot z_i\right) - \left(\sum_{j \neq i}^{n} r_j^* \cdot z_j + r_i^2 \cdot z_i\right) = z_{i,2} \cdot z_i \in [-N, N].$$

It follows that $z_i = z_{i,1} \cdot z_{i,2}^{-1} \mod p^*$, where $z_{i,1} \in [-N, N], z_{i,2} \in [1, D]$ as desired. □

Lemma 4.9 (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 $L := LCM\{z_{i,2}\}_{i\in[n]}$. Also suppose that $L > D > 4$ and that $ND^n < p^*$. Then, we additionally have*

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

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

Importantly, when $b = 1$, this says that if $L > D$, then the prover succeeds only with low 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

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

Here, let $b' = \gcd(z_1, \dots, z_{n-1}, L)$. 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

$$\begin{aligned}
&\frac{1}{D} \sum_{\theta \leftarrow [0,D)} \Pr\left[r_1, r_2, \dots, r_{n-1} \leftarrow [0, D) : \sum_{i=1}^{n-1} r_i \cdot z_i = c - z_n\theta \mod L \right] \\
&\quad \le \frac{1}{D} \cdot \left\lceil \frac{D}{b'} \right\rceil \cdot \left(\frac{1}{D} + \frac{b'}{L} \right) \\
&\quad < \frac{1}{D} \cdot \left(\frac{D}{b'} + 1 \right) \cdot \left(\frac{1}{D} + \frac{b'}{L} \right) \\
&\quad = \left(\frac{1}{b'} + \frac{1}{D} \right) \cdot \left(\frac{1}{D} + \frac{b'}{L} \right) \\
&\quad = \frac{1}{L} + \frac{1}{D} \left(\frac{1}{b'} + \frac{b'}{L} + \frac{1}{D} \right)
\end{aligned}$$

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. □

5 Proving Non-native Modulo Arithmetic

Many commit-and-prove protocols are equipped with a corresponding vector commitment scheme where the committed vectors are over $\mathbb{F}_{p^*}^n$, where $\mathbb{F}_{p^*}$ is some fixed prime field (also see remark 3.4.1 for more discussion). Due to this reason these protocols naturally allow for proving modulo relations over the ring $\mathbb{Z}_{p^*}$ for such fixed choice of p^*.

5.1 Non-native Arithmetic

One of main difficulties in many proof systems is to prove arithmetic relations over some modulus $q \neq p^*$, which is an example of proving non-native arithmetic relations.

Many solutions have been proposed to deal with this difficulty. Some utilize bit decomposition, such as [KPS18]. This approach introduces a large number of constraints (thus jeopardize efficiency) whenever the modulus q becomes large. Other constructions utilize certain special ring encodings, such as [GNSV21]. But this approach only achieves designated-verification (which is less desirable than public verifiablity) and still poses many constraints onto the backend.

One of our main contribution is to propose an efficient solution to this problem for any (reasonably large) modulus $q << p^*$. Our solution is based on a different characterization of $\mathbb{Z}_q$ elements and makes use of the Batch-PoSO protocol described in previous section. Furthermore it builds directly on top of these existing commit-and-prove protocols with minimal overhead.

Jumping ahead out main use case of this technique is for proving $\mathsf{R1CS}$ relations which is defined over some arbitrary ring $\mathbb{Z}_q$. This can be seen as en example of proving modulo q arithmetics. Let's denote such instance by $\mathbb{X}^q = (\mathbb{Z}_q, A, B, C, \text{io}, m, n)$. We thus restrict our attention to commit-and-prove protocols which targets for $\mathsf{R1CS}$ relations. Below we first summarize all our ingredients.

Ingredients.

- A vector commitment scheme $(\mathsf{KeyGen}, \mathsf{Commit})$ over the message space $\mathbb{Z}_{p^*}^n$.
- A commit-and-prove protocol $(\mathsf{KeyGen}, \mathsf{Commit}, \mathcal{P}, \mathcal{V})$ for the vector commitment scheme $(\mathsf{KeyGen}, \mathsf{Commit})$ that allows for proving $\mathsf{R1CS}$ relations over $\mathbb{Z}_{p^*}^n$. We also require it to be an argument of knowledge.

5.2 Overview

We present an high-level overview of our techniques. The most intuitive idea (despite having some challenges) is to transform the $\mathsf{R1CS}$ instance $\mathbb{X}^q$ into a new instance $\mathbb{X}^{p^*}$ such that it is defined over $\mathbb{Z}_{p^*}$. The resulting instance can then be proved via the commit-and-prove protocol which supports proving $\mathbb{Z}_{p^*}$ relations.

For this purpose, we build the following $\mathsf{R1CSTransformer}$: The transformer will perform the following changes to each constraint in $\mathbb{X}^q$. For example, suppose we want the prover to prove the constraint that $a \cdot b = c \mod q$. Instead of proving this relation over modulo q, let's now ask the prover to supply an additional shift value k and subsequently prove that $a \cdot b = c + k \cdot q$ (over $\mathbb{Z}$) , which is a relation that holds over the integers. We then add this constraint to $\mathbb{X}^{p^*}$. In other words, we add the constraint that $a \cdot b = c + k \cdot q$ in the instance $\mathbb{X}^{p^*}$. Nevertheless, since all of (a, b, c, k) now lives in the ring $\mathbb{Z}_{p^*}$, this constraint only holds over $\mathbb{Z}_{p^*}$. Importantly, it does not necessarily imply that $a \cdot b = c \mod q$.

To deal with this issue, we will in fact use a different encoding of $\mathbb{Z}_q$ elements and embed them in $\mathbb{Z}_{p^*}$. We emphasize that this encoding is only possible whenever $q << p^*$. More details follow in subsequent sections.

5.3 Encoding $\mathbb{Z}_q$ elements

Again we will use rational representative as discussed in section 4.2 to encode $\mathbb{Z}_q$ elements. That is, for any value $x \in \mathbb{Z}_q$, it is represented by some rational $\frac{n}{d} \in \mathbb{Q}$ such that $x = n \cdot d^{-1} \mod q$. However, a new issue arises in this setting: Since q can be arbitrary, it might be any composite number. Thus the denominator d might not even admit its inverse modulo q if $\gcd(d, q) \neq 1$. For this reason, we will restrict the set of rational representative to those whose denominator admits inverse modulo q. Similar to definition 4.2, we define the notion of $q-$invertible rational representative:

Definition 5.1 ($q-$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 $x = n \cdot d^{-1} \mod q$.

5.4 Proving Arithmetic Relation Over $\mathbb{Z}_q$

Equipped with such encoding of $\mathbb{Z}_q$ elements, we now explain how to prove any arithmetic relations modulo q. Recall that we add the constraint that $a \cdot b = c + k \cdot q \mod p^*$. Since this constraint is defined over $\mathbb{Z}_{p^*}$, we can assume that each of (a, b, c, k) lives in the ring $\mathbb{Z}_{p^*}$.

We want to embed the ring $\mathbb{Z}_q$ inside $\mathbb{Z}_{p^*}$ as follows: Instead of treating (a, b, c, k) as $\mathbb{Z}_{p^*}$ elements, we will view each as an element of $\mathbb{Z}_q$ (via our $\mathbb{Z}_q$ encodings). To achieve this, we first map $\mathbb{Z}_{p^*}$ elements to their bounded rational representatives: For example, consider the representation of a as $\frac{a_1}{a_2}$ such that ($\gcd(a_1, a_2) = 1, a = \frac{a_1}{a_2} \mod p^*$; a_1, a_2 are both small). We assert that all of (a, b, c, k) admit such bounded representatives. Then we have:

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

Similar to what we have proved in lemma 4.4, if all the rational representative in the above equation have both bounded numerator and bounded denominator, then in fact we can argue that this equation indeed holds over the field of rational numbers, instead of merely modulo p^*.

More concretely, let's assume that all of (a, b, c, k) admit bounded representative in $\mathbb{Q}_{N,D}$ such that $(ND)^2 < q$ and $N \cdot D^3 \cdot (q + 1) < p^*$. Then let's multiply both sides of the equation by the LCM of the denominators:

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

Since $a_1, b_1 \leq N$ and $c_2, k_2 \leq D$, it holds that the left hand side $a_1 \cdot b_1 \cdot c_2 \cdot k_2 < p^*$ and furthermore the right hand side $a_2 \cdot b_2 \cdot c_1 \cdot k_2 + k_1 \cdot a_2 \cdot b_2 \cdot c_2 \cdot q < p^*$. Dividing back the LCM, the following equation will hold over rational numbers:

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

We furthermore consider another mapping which maps each of above rational representative into a $\mathbb{Z}_q$ element: Let's first assert that all these rational representatives $(\frac{a_1}{a_2}, \frac{b_1}{b_2}, \frac{c_1}{c_2}, \frac{k_1}{k_2})$ are also $q-$invertible, then we map them to $\mathbb{Z}_q$ elements by taking the above equation modulo q, which now becomes:

$$\frac{a_1}{a_2} \cdot \frac{b_1}{b_2} = \frac{c_1}{c_2} \mod q$$

Let's denote by $a' = \frac{a_1}{a_2} \mod q$, $b' = \frac{b_1}{b_2} \mod q$, $c' = \frac{c_1}{c_2} \mod q$. We say that $(a', b', c' \in \mathbb{Z}_q)$ are embedded $\mathbb{Z}_q$ elements of $(a, b, c \in \mathbb{Z}_{p^*})$.

Moreover, since $ND < \sqrt{q} < p/2$, by lemma 4.4 (a', b', c') are in fact unique embeddings of (a, b, c). Thus the constraint $a \cdot b = c + k \cdot q \mod p^*$ now implies $a' \cdot b' = c' \mod q$.

To conclude, whenever $(a, b, c) \in \mathbb{Z}_{p^*}$ admit rational representative in $\mathbb{Q}_{N,D} \cap \mathbb{Q}_q$, we can define a mapping $\phi : \{a, b, c\} \to \mathbb{Z}_q$ as follows: Let $\frac{a_1}{a_2} \in \mathbb{Q}_{N,D} \cap \mathbb{Q}_q$ be the rational representative of a, then $\phi(a) \to a'$ where $a' = \frac{a_1}{a_2} \mod q$ (similar for b, c). The mapping ϕ is somewhat homomorphic in that $a \cdot b = c \mod p^*$ implies $a' \cdot b' = c' \mod q$.

5.4.1 Batch-PoSO for Representative in $\mathbb{Q}_{N,D} \cap \mathbb{Q}_q$

One important aspect behind the embedding is that we must ensure an element $a \in \mathbb{Z}_{p^*}$ admit representative in $\mathbb{Q}_{N,D} \cap \mathbb{Q}_q$. In this section we argue that the Batch-PoSO protocol already suffices to prove this by simply tuning the parameters.

Let $q_{\min}$ be the smallest divisor of q. Observe that whenever $D < q_{\min}$, we must have $\mathbb{Q}_{N,D} \subseteq \mathbb{Q}_q$. This is to say $\mathbb{Q}_{N,D} \cap \mathbb{Q}_q = \mathbb{Q}_{N,D}$.

We have shown via lemma 4.8 that the Batch-PoSO protocol as described in Fig 2 guarantees that a vector $\mathbf{z}$ admits rational representative in $\mathbb{Q}_{N,D}$. In order to design a Batch-PoSO for Representative in $\mathbb{Q}_{N,D} \cap \mathbb{Q}_q$, it suffices to update $D = \min(D,\ q_{\min} - 1)$.

5.5 Commit-and-Prove Construction

In this section we present a formal construction of a commit-and-prove protocol which supports proving modulo arithmetics where the modulus $q << p^*$. We start by describing the R1CSTransformer.

The R1CS Transformer $\mathbb{Z}_q \to \mathbb{Z}_{p^*}$ In Fig 3 we give the description of a R1CSTransformer which compiles the R1CS instance $\mathbb{X}^q$ into a new instance $\mathbb{X}^{p^*}$ which is defined over $\mathbb{Z}_{p^*}$. For ease of subsequent security analysis, we further assume that $\mathbb{X}^q$ is k-bounded (see def 3.1 for its definition).

In Fig 4, we present our full commit-and-prove protocol for proving arithmetic relations over $\mathbb{Z}_q$. It uses the following components:

- The R1CSTransformer, in Fig 3.
- The aforementioned ingredients in Sec 5.1.
- The Batch-PoSO protocol in Fig 2.

<u>R1CS transformer which compiles $\mathbb{X}^q$ to $\mathbb{X}^{p^*}$</u>

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

Figure 3: Description of R1CS Transformer

Commit-and-prove for arithmetic over $\mathbb{Z}_q$

1. The prover and the verifier run $\mathsf{R1CSTransformer}$ on the instance $\mathbb{X}^q$ to obtain a $\mathsf{R1CS}$ instance $\mathbb{X}^{p^*} = (\mathbb{Z}_{p^*}, A, B, C, \mathrm{io}, m, (n+m))$.
2. Let $\mathbb{W}$ be prover's witness. The prover computes a shift vector $\mathbf{k} \in \mathbb{Z}_q^m$ such that $(A \cdot \mathbf{z}||\mathbf{k}) \circ (B \cdot \mathbf{z}||\mathbf{k}) = (C \cdot \mathbf{z}||\mathbf{k})$ holds over the integers, where $\mathbf{z} = (1, \mathrm{io}, \mathbb{W}) \in \mathbb{Z}_q^{n+1}$.
3. The prover then sets the final witness value as $\mathbb{W} = \mathbb{W}||\mathbf{k}$. The prover generates $c := \mathsf{Commit}(\mathbb{W}; u)$ with some randomness u, and sends c to the verifier.
4. The prover and verifier execute the commit-and-prove protocol with respect to the $\mathsf{R1CS}$ instance $\mathbb{X}^{p^*}$ and commitment c.
$$\mathcal{P}\left(1^\lambda, (\mathbb{X}^{p^*}, u)\right) \leftrightarrow \mathcal{V}(1^\lambda, \mathbb{X}^p).$$
5. Let $q_{\min}$ be the smallest divisor of q. The prover and verifier execute the Batch-PoSO protocol for the language $(L_{R,1}, L_{N,D})$ on the commitment c, where the parameters are set to be $(R = q,\ D = q_{\min} - 1,\ N = R \cdot D \cdot (m+n))$. This protocol is repeated for $\lambda / \log(D)$ times.
6. The verifier accepts if the prover succeeds in all protocols.

Figure 4: Description of Commit-and-prove for $\mathbb{X}^q$.

Remark 5.2 (Merging Batch-PoSO). *We point out that although there exists two commit-and-prove schemes in Fig 4, one for the instance $\mathbb{X}^{p^*}$ and one for multiple executions of Batch-*PoSO *protocols, these two schemes can be easily merged together by augmenting the instance $\mathbb{X}^{p^*}$ to include all constraints as required for* PoSO *(which itself is also an* $\mathsf{R1CS}$ *instance). We defer the detail of this merging to our final protocol (see Fig 6).*

5.6 Security Proofs for Commit-and-Prove Protocol of Modulo Arithmetics

In this section we give security proofs for our commit-and-prove protocol which is described in fig 4.

Completeness We show that this construction achieves completeness. In particular, if the instance $\mathbb{X}^q$ is satisfiable and the prover $\mathcal{P}$ holds a satisfying witness $\mathbb{W}$ for the instance $\mathbb{X}^q$, then she can convince the verifier $\mathcal{V}$ with probability 1.

The prover will first use $\mathbb{W}$ to compute the 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 $\mathbb{X}^{p^*}$. Therefore the prover can succeed in the commit-and-prove protocol for this instance with probability 1 due to the completeness of the underlying commit-and-prove protocol.

Furthermore, since $(\mathbb{W}||\mathbf{k}) \in \mathbb{Z}_q^{(m+n)}$, due to the completeness of Batch-PoSO (lemma 4.7), the prover will also pass all Batch-PoSO protocols with probability

1.

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 $1/2^\lambda$.

Suppose that $\mathbb{X}^q$ is not satisfiable, we consider two cases:

- **Case I:** Assume the prover's concatenated witness $\mathbb{W}||\mathbf{k}$ does not admit representative in $\mathbb{Q}_{N,D}$. Then by soundness of Batch-PoSO (lemma 4.8), 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 $\mathbb{W}||\mathbf{k}$ admits representative in $\mathbb{Q}_{N,D}$. We argue that If the prover can pass the commit-and-prove protocol for $\mathbb{X}^{p^*}$ with probability greater than $1/2^\lambda$, then $\mathbb{X}^q$ must be satisfiable.

  Consider each constraint $i \in [m]$ in $\mathbb{X}^q$:

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

  This constraint is substituted with the following constraint in $\mathbb{X}^{p^*}$:

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

  Suppose the prover succeeds in the commit-and-prove protocol. Due to knowledge soundness of the commit-and-prove protocol, we can extract from the prover the extended witness vector $\mathbf{z}||\mathbf{k} \in \mathbb{Z}_q^{n+m}$ such that $\mathbf{z}$, $\mathbf{k}[i]$ will pass the above constraint. Since the prover's witness $\mathbb{W}||\mathbf{k}$ admits representative in $\mathbb{Q}_{N,D}$, 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 representative for $\mathbf{z}$ (and similarly for $\mathbf{k}$), then we have:

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

  Multiply by LCM of denominators (denote the LCM by L), we have:

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

  Since $\mathbb{X}^q$ is k-bounded, the value of $A[i] \cdot \mathbf{z_1} \cdot B[i] \cdot \mathbf{z_1}$ is at most k assuming $\mathbf{z_1} \in \mathbb{Z}_k^{n+1}$. However, the guarantee that we have on $\mathbf{z}$ is that $\mathbf{z_1} \in \mathbb{Z}_{N=q \cdot D \cdot (m+n)}^{n+1}$. Nonetheless under this condition it will blow up the value of $A[i] \cdot \mathbf{z_1} \cdot B[i] \cdot \mathbf{z_1}$ by at most $(N/k)^2$. 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 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 lemma 4.9, the LCM L is at most D. Since $k \cdot D^2 \cdot (q \cdot (m+n))^2 < k \cdot q_{\min}^2 \cdot (q \cdot (m+n))^2 < p^*$, this equation must also hold over the integers. Thus we have:

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

Moreover, since we have set $D = q_{\min} - 1$, as discussed in Sec 5.4.1, this further implies that both $\mathbf{z}$ and $\mathbf{k}[i]$ admit representative in $\mathbb{Q}_q$ (thus can be viewed as element in $\mathbb{Z}_q$). Finally we define the corresponding embedded $\mathbb{Z}_q$ elements as $\mathbf{z}' = \frac{\mathbf{z_1}}{\mathbf{z_2}} \mod q$. It can be seen that:

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

Thus $\mathbf{z}'$ contains a satisfying witness for the instance $\mathbb{X}^q$, and this is a contradiction.

6 Improved Frontend for SIMD Computations

In this section we show how to build a commit-and-prove protocol for any computation that is *data-parallel* (SIMD) (see def 3.7.1). We point out that our core effort is to design a better frontend (an compiler to R1CS instances) which can be readily combined with many existing commit-and-prove schemes (backends).

We first list the necessary ingredients as follows.

Ingredients.

- A commit-and-prove protocol for arithmetics over $\mathbb{Z}_q$ for any choice of q as shown in Fig 3;
- A *k-bounded* and *p-satisfiable* EQC compiler which takes as input some program G, and outputs some R1CS instance. (see Sec 3.6.1);
- A CRT packing scheme (see Def 4.1).

Roadmap: Our roadmap in this section is as follows: The primary goal is to design a specific R1CSCompiler that outputs an optimized R1CS instance for any SIMD computation. We will start with a toy example in Sec 6.1 where we will explain the idea behind such a compiler for a simple SIMD computation. We will then extend this idea to general SIMD computation in Sec 6.2. Then in Sec 6.3 we will present the full description of our specific R1CSCompiler. We then combine this compiler with the commit-and-prove protocol for arithmetics over $\mathbb{Z}_q$ described in the previous section so as to derive an optimized commit-and-prove protocol for SIMD computation.

6.1 A Toy Example

Consider a simple SIMD computation in the form of a circuit C which consists of two copies of the subcircuit G. The subcircuit G takes a single input x and outputs $G(x) = x(x+1)$ (where the operation is over the integers). That is $C(x_1, x_2) := (G(x_1), G(x_2)) = (x_1(x_1+1), x_2(x_2+1))$. Let's furthermore denote this pair of output values as (y_1, y_2).

Observe that in order to express the correctness of computing circuit C, we could simply enforce the following two constraints:

$$y_1 = x_1(x_1 + 1);\ y_2 = x_2(x_2 + 1).$$

Our improvement stems from the following goal: We would like to combine these two constraints into one single constraint. In order to achieve this, notice that although both constraints are defined over the integers, whenever the values of x_1 and x_2 are small, we can in fact consider these constraints over finite rings. For example: Assume that $x_1 \leq \sqrt{q_1} - 1$ for some prime q_1, then we have $x_1(x_1+1) < q_1$. Consider the ring $\mathbb{Z}_{q_1}$, observe that

$$y_1 = x_1(x_1+1) \mod q_1 \iff y_1 = x_1(x_1+1).$$

Similarly, let's pick a different prime number q_2 such that $x_2 \leq \sqrt{q_2} - 1$, then we can instead write these two constraints:

$$y_1 = x_1(x_1+1) \mod q_1;\ y_2 = x_2(x_2+1) \mod q_2.$$

Notice that both constraints are expressed as finite ring operations (over $\mathbb{Z}_{q_1}$ and $\mathbb{Z}_{q_2}$ respectively). This readily suggests a way of packing those two constraints using the CRT isomorphism.

Applying the CRT packing (see Sec 4.1), let $q = q_1 \cdot q_2$. We pack the values into $X = \mathsf{CRT.Pack}(x_1, x_2)$, $Y = \mathsf{CRT.Pack}(y_1, y_2)$. The ring isomorphism $\mathbb{Z}_q \cong \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_1}$ implies that:

$$Y = X(X+1) \mod q \iff y_1 = x_1(x_1+1) \wedge y_2 = x_2(x_2+1).$$

Therefore, instead of writing two constraints, we could have merged them into a single constraint of the same format. This comes at the cost of packing the values inside a larger modulus.

6.2 Handling General SIMD Computations

In general, we can consider any SIMD computation C where the sub-computation G is repeated ℓ times with ℓ different independent inputs $(x_1, \ldots, x_\ell)$. That is: $C(x_1, \ldots, x_\ell) = (G(x_1), \ldots, G(x_\ell))$.

We first apply the *k-bounded* and *p-satisfiable* EQC compiler on the program G. The compiler outputs some $\mathsf{R1CS}$ instance which we shall denote by $\mathbb{X}_G = (A_G, B_G, C_G, \mathrm{io} = \bot, m, n)$. For each sub-computation $i \in [\ell]$, let y_i be its alleged output. We denote by $\mathbb{X}_G^i$ where we substitute the input/output values io_i with the corresponding value (x_i, y_i): That is, $\mathrm{io}_i = x_i || y_i$. It follows that $\mathbb{X}_G^i$ encodes the satisfiability of the sub-computation G on input value x_i and output y_i.

Let $\mathbb{W}_i$ be prover's alleged witness for the i^{th} instance. Let $\mathbf{z_i} = (1, \mathrm{io}_i, \mathbb{W}_i) \in \mathbb{Z}_k^{n+1}$ be its extended witness vector.

We want to use CRT packing to pack all the ℓ instances into just one instance. First, we choose ℓ smallest different prime numbers $(q_1, \ldots, q_\ell)$ such that each $q_i \geq p$. Let $q = \prod_{i=1}^{\ell} q_i$ and let $\mathbf{z} = \mathsf{CRT.Pack}(\mathbf{z_1}, \ldots, \mathbf{z_\ell})$. Observe that the CRT isomorphism implies that:

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

$$\Updownarrow$$

$$\forall i \in [\ell] :\ (A_G \cdot \mathbf{z_i}) \circ (B_G \cdot \mathbf{z_i}) = C_G \cdot \mathbf{z_i} \mod q_i.$$

Due to *p-satisfiability* of the compiler, it further holds that:

$$\forall i \in [\ell]: \quad (A_G \cdot \mathbf{z_i}) \circ (B_G \cdot \mathbf{z_i}) = C_G \cdot \mathbf{z_i} \mod q_i$$
$$\Updownarrow$$
$$(A_G \cdot \mathbf{z_i}) \circ (B_G \cdot \mathbf{z_i}) = C_G \cdot \mathbf{z_i} \mod p.$$

To conclude, let io $= \mathsf{CRT.Pack}(\mathrm{io}_\mathbf{1}, \ldots, \mathrm{io}_\ell)$ and let the packed $\mathsf{R1CS}$ instance be $\mathbb{X}_C = (\mathbb{Z}_q, A_G, B_G, C_G, \mathrm{io}, m, n)$. It is satisfiable if and only if all of $\left\{\mathbb{X}_G^i\right\}_{i\in[\ell]}$ are satisfiable. In other words, $\mathbb{X}_C$ encodes the satisfiability of the SIMD circuit C.

The only caveat now is that the instance $\mathbb{X}_C$ is defined over some arbitrarily chosen modulus q. Furthermore, the modulus q is very large (this increases with the number of sub-computations).

Fortunately, we already know how to deal with this issue: The aforementioned commit-and-prove protocol for modulo arithmetics over $\mathbb{Z}_q$ can be used to prove such instance.

6.3 Commit-and-Prove Protocol for SIMD Computation

In this section we formally present our commit-and-prove protocol for any SIMD computation C. Let C consists of N identical copies of the sub-computation G, each copy i has its input and alleged output io_i. For some choice of ℓ (which we refer to as the packing factor from now on), We will split these N copies into N/ℓ batches, each batch consisting of ℓ copies. The description of $\mathsf{R1CSCompiler}$ is given in Fig 5.

<u>R1CS Compiler for SIMD computations</u>

1. Apply the *k-bounded* and *p-satisfiable* EQC compiler on the program G, and let $\mathbb{X}_G = (A_G, B_G, C_G, \mathrm{io} = \perp, m, n)$ be the resulting $\mathsf{R1CS}$ instance for G.
2. Choose ℓ smallest different prime numbers $(q_1, \ldots, q_\ell)$ such that each $q_i \geq p$. Let $q = \prod_{i=1}^{\ell} q_i$ and assert that $k \cdot p^2 \cdot (q \cdot (m+n))^2 < p^*$.
3. Let $T = N/\ell$. For each $j \in [T]$, do the following:
   - For $i \in [\ell]$, let io_i^j be the input and alleged output for the $(j \cdot T + i)^{\text{th}}$ sub-computation.
   - Pack the values $\{\mathrm{io}_i^j\}_{i\in[\ell]}$ via CRT packing: $\mathsf{CRT.Pack}(\{\mathrm{io}_i^j\}_{i\in[\ell]}) \to \mathrm{io}^j$.

   Construct the following matrices:
   $$A_C = \mathsf{diag}^T(A_G);$$
   $$B_C = \mathsf{diag}^T(B_G);$$
   $$C_C = \mathsf{diag}^T(C_G).$$
4. Set $\mathbb{X}_C = (\mathbb{Z}_q, A_C, B_C, C_C, \mathrm{io}^1 || \ldots || \mathrm{io}^T, m * T, n * T)$.

Figure 5: Description of $\mathsf{R1CS}$ compiler for SIMD computations.

After obtaining the $\mathsf{R1CS}$ instance $\mathbb{X}_C$ from the compiler, we then use our commit-and-prove protocol for arithmetics over $\mathbb{Z}_q$ (described in Fig 4) to prove such instance. This concludes our commit-and-prove protocol for SIMD computations.

For ease of subsequent analysis, we also provide a complete description of our commit-and-prove protocol in Fig 6. Notice that in the final protocol, we only make a single black-box use of the underlying commit-and-prove construction. In other words, our contribution can be seen as an improved frontend for SIMD computations.

A Commit-and-Prove Protocol for SIMD Computations

- **Basic Ingredients**:
  - A commit-and-prove protocol for proving arithmetic relations over $\mathbb{Z}_{p^*}$ where p^* is a fixed prime determined by the underlying vector commitment scheme (see Def 3.2);
  - A *k-bounded* and *p-satisfiable* EQC compiler which takes as input any program G, and outputs some $\mathsf{R1CS}$ instance. (see Sec 3.6.1);
  - A CRT packing scheme (see Def 4.1).
- **Preliminaries:**
  - Let SIMD computation C consist of N identical copies of the sub-computation G, where each copy i has its input and alleged output as io_i.
  - Let ℓ be the packing factor, we will split these N copies into N/ℓ batches, each batch consisting of ℓ copies.

Full Construction

- **Preprocessing Phase:**
  1. Apply the EQC compiler to the program G, and let $\mathbb{X}_G = (A_G, B_G, C_G, \mathrm{io} = \perp, m, n)$ be the resulting $\mathsf{R1CS}$ instance for G.
  2. Choose ℓ smallest different prime numbers $(q_1, \ldots, q_\ell)$ such that each $q_i \geq p$. Let $q = \prod_{i=1}^{\ell} q_i$ and assert that $k \cdot p^2 \cdot (q \cdot (m+n))^2 < p^*$.
  3. Let $T = N/\ell$. For each $j \in [T]$, do the following:
     - For $i \in [\ell]$, let io_i^j be the input and alleged output for the $(j \cdot T + i)^{\mathrm{th}}$ sub-computation.
     - Pack the values $\{\mathrm{io}_i^j\}_{i\in[\ell]}$ via CRT packing: $\mathsf{CRT.Pack}(\{\mathrm{io}_i^j\}_{i\in[\ell]}) \to \mathrm{io}^j$.

     Construct the following matrices:
     $$A_C = \mathsf{diag}^T(A_G);$$
     $$B_C = \mathsf{diag}^T(B_G);$$
     $$C_C = \mathsf{diag}^T(C_G).$$
  4. Set merged $\mathsf{R1CS}$ instance to be $\mathbb{X}_C^q = (\mathbb{Z}_q, A_C, B_C, C_C, \mathrm{io}^1||\ldots||\mathrm{io}^T, m * T, n * T)$.
  5. Apply $\mathsf{R1CSTransformer}$ (described in Fig 3) to the instance $\mathbb{X}_C^q$ to obtain a $\mathsf{R1CS}$ instance $\mathbb{X}_C^{p^*} = (\mathbb{Z}_{p^*}, A'_C, B'_C, C'_C, \mathrm{io}^1||\ldots||\mathrm{io}^T, m * T, (m+n) * T)$.

A Commit-and-Prove Protocol ... (continued)

- **Commit-and-Prove Phase:**
  1. Let $\{\mathbb{W}_i^j\}_{i\in[\ell],j\in[T]}$ be the set of corresponding witness values. The prover set $\mathbf{z_i^j} = (1, \mathrm{io}_i^j, \mathbb{W}_i^j)$ and then pack these values as $\mathbf{z^j} = \mathsf{CRT.Pack}(\mathbf{z_1^j}, \ldots, \mathbf{z_\ell^j})$ for each $j \in [T]$. Finally it sets the concatenated extended witness as $\mathbf{z} = \mathbf{z^1}|| \ldots ||\mathbf{z^T}$.
  2. The prover then computes the vector $\mathbf{k} \in \mathbb{Z}_q^{m*T}$ such that $(A'_C \cdot \mathbf{z}||\mathbf{k}) \circ (B'_C \cdot \mathbf{z}||\mathbf{k}) = (C'_C \cdot \mathbf{z}||\mathbf{k})$ over the integers.
  3. The prover then resets the extended witness as $\mathbf{z} = \mathbf{z}||\mathbf{k}$ and commits to it as $c := \mathsf{Commit}(\mathbf{z}||\mathbf{k}; u)$ with some randomness u, and then sends c to the verifier.
  4. **Merging $\lambda/\log(p)$ executions of Batch-PoSO:**
     (a) Both prover and verifier set up the following PoSO parameters: $(R = q,\ D = p,\ N = R \cdot p \cdot (m+n) * T)$.
     (b) For $j \in [\lambda/\log(p)]$, the verifier samples random vector $\mathbf{r}^j \in \mathbb{Z}_{p^*}^{(m+n)*T}$ such that each entry $\mathbf{r}^j[i]$ is small (i.e. $\forall i \in [(m+n)*T], \mathbf{r}^j[i] \in [0, D)$). It then sends them to the prover, who computes $v^j :=< \mathbf{z} \cdot \mathbf{r}^j >$
     (c) The prover and verifier prepare the following R1CS instance: $\mathbb{X}_{\mathsf{PoSO}} = (A, B, C, \mathrm{io} = \perp, \lambda/\log(p), (m+n)*T+1)$, 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 $C = \begin{bmatrix} 0\ldots||1||0 \\ \ldots \\ 0\ldots||1||0 \end{bmatrix}$.
     (d) Both the prover and verifier augment the R1CS instance $\mathbb{X}_C^{p^*}$ with $\mathbb{X}_{\mathsf{PoSO}}$. Let's denote this final instance as $\mathbb{X}^*$.
     (e) The prover sets $\mathbf{v} = v^1|| \ldots ||v^{\lambda/\log(p)}$, and resets the extended witness as $\mathbf{z} = \mathbf{z}||\mathbf{v}$, where $\mathbf{v}$ is part of the output (i.e. it appears in io). It then sends $\mathbf{v}$ as part of the claimed output to the verifier.
  5. The prover and verifier execute the commit-and-prove protocol with respect to the R1CS instance $\mathbb{X}^*$ and commitment c.
  $$\mathcal{P}\left(1^\lambda, (\mathbb{X}^*, u)\right) \leftrightarrow \mathcal{V}(1^\lambda, \mathbb{X}^*).$$
  6. If this passes and that $\mathbf{v}[j] \in [N]$ for each $j \in [|\mathbf{v}|]$, the verifier accepts, otherwise it rejects.

Figure 6: Description of our full commit-and-prove protocol for SIMD computations.

6.4 Security Proofs for Commit-and-Prove Protocol for SIMD Computation

In this section we give security proofs of our commit-and-prove protocol for SIMD computation described in Fig 6.

Completeness We show that this construction achieves completeness. In particular, suppose the computation C is correct, that is, the alleged outputs is the result of applying the computation C over its inputs. In this case we argue that the prover can pass the commit-and-prove protocol with probability 1.

Since the computation C is correct, each of sub-computation G must be correct with respect to the corresponding inputs and outputs in $\{\mathrm{io}_i^j\}_{i\in[\ell],j\in[T]}$. Therefore, we can find the corresponding witness such that each $\mathsf{R1CS}$ instance $\mathbb{X}_G = (A_G, B_G, C_G, \mathrm{io}_i^j, m, n)$ is satisfiable. Let $\{\mathbb{W}_i^j\}_{i\in[\ell],j\in[T]}$ be the set of corresponding witness values. Define $\mathbf{z_i^j} = (1, \mathrm{io}_i^j, \mathbb{W}_i^j)$. Then it holds that for all $i \in [\ell]$ and $j \in [T]$:

$$(A_G \cdot \mathbf{z_i^j}) \circ (B_G \cdot \mathbf{z_i^j}) = C_G \cdot \mathbf{z_i^j} \mod p.$$

Due to *p-satisfiability* of the compiler, we have:

$$(A_G \cdot \mathbf{z_i^j}) \circ (B_G \cdot \mathbf{z_i^j}) = C_G \cdot \mathbf{z_i^j} \mod q_i.$$

Now we define the CRT packed value $\mathbf{z^j} = \mathsf{CRT.Pack}(\mathbf{z_1^j}, \dots, \mathbf{z_\ell^j})$. Due to the CRT isomorphism, it holds that:

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

Thus the concatenated extended witness $\mathbf{z} = \mathbf{z^1}||\dots||\mathbf{z^T}$ must be a satisfying witness for the instance $\mathbb{X}_C$. The prover can thus succeed with probability 1 due to the completeness of our commit-and-prove protocol for arithmetic over $\mathbb{Z}_q$.

Soundness We show that this construction achieves soundness. In particular, suppose that the output of C is not the correct evaluation over its inputs, we argue that in this case the prover can pass the commit-and-prove protocol with probability at most $1/2^\lambda$.

In fact, it suffices to show that the instance $\mathbb{X}_C$ output by our compiler can not be satisfiable. It then follows from our soundness argument in Sec 5.6 that the prover can succeed in the commit-and-prove protocol with probability at most $1/2^\lambda$.

In order to show that the instance $\mathbb{X}_C$ output by our compiler can not be satisfiable, we prove the contra-positive: Suppose $\mathbb{X}_C$ is satisfiable with respect to some witness $\mathbb{W}$, then the output of C must be the correct evaluation over its inputs.

First, we parse the witness as $\mathbb{W} = \mathbb{W}^1||\dots||\mathbb{W}^T$. Furthermore let $\mathbf{z^j} = (1, \mathrm{io}^j, \mathbb{W}^j)$ be the extended witness vector for each $j \in [T]$. Since $\mathbf{z^j}$ is a satisfying witness with respect to $\mathbb{X}_G^j$, we must have:

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

Now let $(\mathbf{z_1^j}, \dots, \mathbf{z_\ell^j}) = \mathsf{CRT.Unpack}(\mathbf{z^j})$. Due to the CRT isomorphism, it holds that for each $i \in [\ell]$:

$$(A_G \cdot \mathbf{z_i^j}) \circ (B_G \cdot \mathbf{z_i^j}) = C_G \cdot \mathbf{z_i^j} \mod q_i.$$

Furthermore, due to *p-satisfiability* of the compiler, we have:

$$(A_G \cdot \mathbf{z_i^j}) \circ (B_G \cdot \mathbf{z_i^j}) = C_G \cdot \mathbf{z_i^j} \mod p.$$

Thus each instance $\mathbb{X}_G$ is satisfiable with respect to inputs and outputs io_i^j. Therefore, for each sub-computation G, the output is the correct evaluation over its input. Thus the output of C must be the correct evaluation over its inputs.

7 Extensions and Optimizations

We will describe some useful extensions and optimizations to our above commit-and-prove protocol for SIMD computations in this section.

7.1 Adding Succinct Verification

The aforementioned commit-and-prove protocol for SIMD computations is mainly designed to optimize prover efficiency. Nevertheless, it incurs a large cost on the verifier due to the use of Batch-PoSO protocol, hence it does not meet succinct verification. More precisely, the verifier's work (running time) in this protocol is linear in the length of the prover's witness (which is comparable to the size of the instance). This is undesirable in many practical use cases where we demand succinct verification.

We describe in this section how to make the verification succinct (assuming the underlying commit-and-prove protocol has succinctness). Let's first identify the verifier's work in the Batch-PoSO protocol implicitly defined in Fig 6:

1. Let's denote the number of executions of Batch-PoSO by K. In this case $K = \lambda / \log(p)$. The verifier first needs to sample K random vectors, each of which has length $(m + n) * T$.
2. The verifier needs to read the value $\mathbf{v}$ which is the result of K Batch-PoSO executions, and furthermore checks that $\mathbf{v}$ has small entries (i.e. smaller than N).
3. The verifier needs to merge the two $\mathsf{R1CS}$ instances $\mathbb{X}_C^{p^*}$ and $\mathbb{X}_{\mathsf{PoSO}}$. The first instance has size (in terms of dimension) $(m * T, (m + n) * T)$ and the second instance has size $(K, (m + n) * T)$.

Our goal is to make the verification independent of (m, n, T). Instead it should only dependent on the security parameter λ. That is, the verifier's runtime should be $p(\lambda)$ for some apriori fixed polynomial $p(\cdot)$.

We now explain how to modify our protocol so as to achieve $O(p(\lambda))$ verification time. In particular, we will modify the above three tasks one by one:

1. **Reusing Randomness:** We notice that the reason that verifier needs to sample random vectors of length $(m + n) * T$ is that in the Batch-PoSO protocol, the prover holds a vector $\mathbf{z}$ of length $(m + n) * T$. To deal with this problem, we make the following changes: Instead of executing a single Batch-PoSO protocol on the vector $\mathbf{z}$, we will break $\mathbf{z}$ into $c = \frac{(m+n)*T}{p(\lambda)}$ number of chunks, each chunk with size $p(\lambda)$. Let's split it into $\mathbf{z} = \mathbf{z_1} || \dots \mathbf{z_c}$, where each $\mathbf{z_i}$ has length $p(\lambda)$. Then we execute a Batch-PoSO protocol for each chunk $\mathbf{z_i}$. Importantly, we can ask the verifier to sample only one random vector $\mathbf{r} \in \mathbb{Z}_D^{p(\lambda)}$, and reuses 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. **Proving Smallness Inside $\mathsf{R1CS}$:** Recall that the second task for verifier is to check that the vector $\mathbf{v}$ has small entries, where the length of this vector depends on the number of Batch-PoSO. However, after reusing the randomness, we have a total number of $c \cdot K = O(\frac{(m+n)*T}{p(\lambda)})$ Batch-PoSO, thus making this vector's size depend on (m, n, T). To remove this overhead, we will instead ask the prover to show that $\mathbf{v}$ has small entries inside the $\mathsf{R1CS}$ instance (so that the verifier does not need to check for smallness). This can be easily achieved via the bit-decomposition method, as follows:

   (a) For each entry $\mathbf{v}[i]$, we bit-decompose this value into $b_1, \ldots, b_{\log(N)}$ number of bits.

   (b) Add the $\mathsf{R1CS}$ constraint that $b_j^2 - b_j = 0$ for each bit b_j, which enforces b_j take binary values.

   (c) Add the $\mathsf{R1CS}$ constraint that $\mathbf{v}[i] = \sum_{j=1}^{\log(N)} 2^j \cdot b_j$, which shows the correct bit-decomposition.

   Notice that these two constraints ensure that the entry $\mathbf{v}[i] \leq N$. One subtle issue with this approach is that the prover's extended witness becomes $\mathbf{z}||\mathbf{v}||\mathbf{b}$ (where $\mathbf{b}$ consists of all bit decompositions), whereas the verifier previously only receives the commitment to $\mathbf{z}$. Therefore we need to allow the prover to update its commitment as follows: The prover initially sends the verifier a commitment to the vector $\mathbf{z}||\underbrace{00\ldots0}_{|\mathbf{v}|+|\mathbf{b}|}$. Then she sends the commitment $\underbrace{00\ldots0}_{|\mathbf{z}|}||\mathbf{v}||\mathbf{b}$ to the verifier upon seeing the randomness. She furthermore proves to the verifier that both commitments are well-formed (i.e. correctly padded with 0's). The verifier can then use the linear homomorphic property of vector commitment (which is already inherent in most SNARKs, also see below for more discussions) to combine these two commitments and retrieve a commitment to $\mathbf{z}||\mathbf{v}||\mathbf{b}$.

3. **Fast Augmentation of $\mathsf{R1CS}$ Instance:** The verifier's third task is to merge the two $\mathsf{R1CS}$ instances $\mathbb{X}_C^{p^*}$ and $\mathbb{X}_{\mathsf{PoSO}}$, both being very large. We observe that in many existing commit-and-prove SNARKs, these instances are typically given as succinct commitments to the verifier so as to achieve holography. Since we are interested in succinct verification, we mainly consider these type of commit-and-prove schemes.

   Whereas the first instance can be given as a commitment value (due to preprocessing) to the verifier, the verifier needs to prepare (and commit to) the second instance, and then merges the two committed instances. While this problem seems to be difficult for any arbitrary commitment scheme, fortunately in practice most commit-and-prove SNARKs utilize a special type of vector commitment scheme that is linearly homomorphic (see Sec 3.2 for more discussion). Furthermore, in many holographic schemes such as [CHM$^+$20] [COS20] [Set20], the commitment of $\mathsf{R1CS}$ 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.

This suggests a way for fast augmentation of $\mathsf{R1CS}$ instances: We will create the final $\mathsf{R1CS}$ instance $\mathbb{X}^*$ as follows:

(a) During the preprocessing phase, we first commit to the positions (indices) of all non-zero entries of both instances $\mathbb{X}_C^{p^*}$ and $\mathbb{X}_{\mathsf{PoSO}}$. Notice that this is doable since $\mathbb{X}_C^{p^*}$ is generated in the preprocessing phase, and furthermore 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 depends on the verifier's randomness. For each repetition $j \in [c \cdot K]$, the non-zero entries have the form $\mathbf{r^j}||\mathbf{r^j}||\ldots||\mathbf{r^j}$ where $\mathbf{r^j} \in \mathbb{Z}_D^{p(\lambda)}$. 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:

$$\begin{aligned}
c_1 &= \mathsf{Commit}([\underbrace{1,0,\ldots,0}_{p(\lambda)},\underbrace{1,0,\ldots,0}_{p(\lambda)},\ldots\underbrace{1,0,\ldots,0}_{p(\lambda)}], r_1),\\
c_2 &= \mathsf{Commit}([\underbrace{0,1,\ldots,0}_{p(\lambda)},\underbrace{0,1,\ldots,0}_{p(\lambda)},\ldots\underbrace{0,1,\ldots,0}_{p(\lambda)}], r_2),\\
&\vdots\\
c_{p(\lambda)} &= \mathsf{Commit}([\underbrace{0,0,\ldots,1}_{p(\lambda)},\underbrace{0,0,\ldots,1}_{p(\lambda)},\ldots\underbrace{0,0,\ldots,1}_{p(\lambda)}], r_{p(\lambda)}).
\end{aligned}$$

Then the verifier can commit to the non-zero entries by computing $c_{\mathbf{r^j}} = \sum_{i=1}^{p(\lambda)} c_i \cdot \mathbf{r^j}[i]$. This takes a total of $p(\lambda)$ operations.

(c) In order to combine the commitments of non-zero entries for all repetitions of PoSO, we can again use the homomorphic property of vector commitments. This step largely resembles the above method, but we will instead treat each committed vector as non-zero entries. Overall, it takes the verifier $O(p(\lambda))$ operations to commit to all the non-zero entries in the instance $\mathbb{X}_{\mathsf{PoSO}}$.

(d) To summarize, the verifier will combine the two instances $\mathbb{X}_C^{p^*}$ and $\mathbb{X}_{\mathsf{PoSO}}$ as follows: It obtains the commitment to indices of all non-zero entries, the commitment to values of non-zero entries in $\mathbb{X}_C^{p^*}$ as well as the $c \cdot K \cdot p(\lambda)$ mask commitments in the preprocessing phase. It will additively combine the masks with its randomness, and then combine it with the commitment of non-zero entries in $\mathbb{X}_C^{p^*}$. Overall, this takes the verifier $O(p(\lambda))$ operations.

7.2 Turning into zk-SNARK

In this section we describe how to further obtain zero-knowledge and moreover turn our scheme into zero-knowledge commit-and-prove SNARK (zk-SNARK).

Honest Verifier Zero-knowledge: We observe that the view of the verifier in our above modified protocol is the same as its view in the final commit-and-prove protocol. Hence if the underlying commit-and-prove protocol satisfies honest verifier zero-knowledge, so does our protocol.

Fiat-Shamir Transform: We can easily turn our protocol into a non-interactive protocol by applying Fiat-Shamir heuristics. Instead of letting the verifier send the randomness, we will generate them using some hash function applied to the prover's commitment c. More specifically, we will read every $\log(D)$ bits of the hash outputs and interpret it as a random value in $\mathbb{Z}_D$. Thus if the underlying commit-and-prove protocol is a zk-SNARK, so will be our protocol.

7.3 Adding Support for Highly Repetitive Computations

We point out that our protocol for SIMD computations can be easily modified so as to support highly repetitive computations. We start with a simple example: Consider the highly repetitive computation C where the sub-computation G is repeated ℓ times iteratively: That is: $C(x) = \underbrace{G(G \ldots G(x))}_{\ell \text{ times}}$.

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 iterative computation of C as consisting of the following two components:

- A SIMD computation C' where the sub-computation G is repeated ℓ times with ℓ different independent inputs $(x_1, \ldots, x_\ell)$. That is: $C(x_1, \ldots, x_\ell) = (y_1 = G(x_1), \ldots, y_\ell = G(x_\ell))$. For this component, we can use our commit-and-prove protocol for SIMD computations. Let ℓ be the packing factor. Recall that the inputs and outputs are packed via CRT as:

$$\mathsf{CRT.Pack}(\{x\}_{i\in[\ell]}) \to x;$$

$$\mathsf{CRT.Pack}(\{y\}_{i\in[\ell]}) \to y.$$

- Constraints which enforces consistency between the current output value y_i and next input value x_{i+1}. However, since both inputs and outputs are given as packed values in the witness, we must first unpack these values in order to complete this constraint. In more details, we will add the following constraints:
  - **Unpacking Constraints:** $x = \mathsf{CRT.Pack}(\{x\}_{i\in[\ell]})$. This is equivalent to adding the single $\mathsf{R1CS}$ constraints that $x = \sum_{i=1}^{\ell} x_i \cdot \lambda_i \mod q$, where λ_i's are apriori fixed Lagrange coefficients as defined in Sec 3.1. Similarly we will add these unpacking constraints for y.

– **Consistency Constraints:** $x_i = y_{i-1}$ for all $i \in [\ell]$.

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

8 Implementations and Evaluations

We first explain the metrics which we will use to measure the performance of SNARK frontend and backend. In out setting, since our compiler outputs an R1CS instance, we will use the complexity of R1CS to serve as metrics for frontend.

8.1 Implementation

We implement the frontend for R1CS in JavaScript that outputs an R1CS instance given the number of repeated copies of SHA-256 programs to be packed, with non-black box usage of `circom`'s code to work with directly with R1CS. `circom` code for SHA-256 was obtained from the `circomlib` package and modified to ensure smallness of all matrix and witness entries. We then implement a modified version of Marlin's commit-and-prove protocol in Rust from the Marlin backend arkworks library [ac22] to benchmark the prover's time as well as memory usage for the circuits output by our frontend implemented above.

We run this code on an Amazon Linux EC2 instance (i4i.8xlarge) with 32 vCPUs and 256GB RAM, due to large memory usage, especially for larger circuit sizes.

Parameter Choices For our experiments, we notice that the SHA256 circuit (with minor modifications) satisfies the *k-bounded* property with $k = 4$ and the output by `circom` (with the optimisation flag `--O1`) satisfies *p-satisfiability* with $p = 2^8$. We can then apply our compiler, choosing the largest set of primes greater than p that satisfy the condition $kp^2(q(m+n)^2) < p^*$. In all our experiments, we have $(m+n) < 2^{20}$ and $p^* \approx 2^{254}$ (we use the curve `bls12-381`). This constrains $q < 2^{93}$, and we can write q as a product of at most 11 primes that are larger than p, and this gives us our maximal packing factor of 11 while obtaining 80-bit security.

8.2 Our Experiments

To estimate prover time for the packed version, we add the additional components of the protocol (updating the verification key) on top of the Marlin prover - these consist of updating the proving key commitment to an LDE of a matrix polynomial of C to contain the random values generated for PoSO. This is relatively lightweight, and the main prover cost comes from the additional non-zero entries and constraints that are added, which are presented above in Table 7. The packed verifier consists of the base verifier and a similar updation procedure for the verification key (which is a fixed cost regardless of the instance size; details in Sec. 7.1). Table 7 contains details of prover time and properties of the R1CS matrices for different instance sizes, and the improvement our frontend provides over using a naïve compiler.

	Packed			
Instance Size	P_{time}(s)	Mem Usage(GB)	Dim.	Non-Zero
11	96	3.46	70702	886336
44	301	12.52	279076	3530461
88	543	24.5	558151	7060911
176	1058	47.15	1116301	14121811

	Baseline			
Instance Size	P_{time}(s)	Mem Usage(GB)	Dim.	Non-Zero
11	188	4.4	321553	1483856
44	688	16.5	1286209	5935424
88	1377	31.8	2572417	11870848
176	2760	63.6	5144833	23741696

Figure 7: (R1CS instance $\mathbb{X} = (A, B, C, \text{io}, m, n)$) Table of values for Prover time, memory usage, R1CS dimension $\max(m, n)$ and number of non-zero entries in A, B, C for our frontend compared to a naïve implementation for n repeated sub-computations of SHA-256 (with 128-bit input size). The instance size is the number of SHA-256 programs that are compiled into the final circuit with our frontend ("packed") or the naïve technique ("baseline"). The packing factor is 11, as computed in Sec. 8.

We obtain a final verification time of approximately 900 milliseconds, most of which is taken up by computing the updated commitment in the verification key, specifically 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 2 evaluations and 2 commitments (group elements) to the proof, which add another 171 bytes to the proof, bringing us to a larger but constant proof size around $1KB$.

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.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, Heidelberg, August 2019.

[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`.

[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 zk-SNARKs 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.

[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.

[PS01] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13, 10 2001.

[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.

[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.