Gemini: Elastic SNARKs for Diverse Environments

Jonathan Bootle[1][0000−0003−3582−3368], Alessandro Chiesa[2,3], Yuncong Hu[3][0000−0002−8338−3507] and Michele Orrù[3][0000−0001−6518−2712]

[1] IBM Research, Zurich, Switzerland
`jbt@zurich.ibm.com`
[2] École polytechnique fédérale de Lausanne, Lausanne, Switzerland
`alessandro.chiesa@epfl.ch`
[3] University of California, Berkeley, Berkeley, USA
`yuncong_hu@berkeley.edu`
`michele.orru@berkeley.edu`

Abstract. We introduce a new class of succinct arguments, that we call elastic. Elastic SNARKs allow the prover to allocate different resources (such as memory and time) depending on the execution environment and the statement to prove. The resulting output is independent of the prover's configuration. To study elastic SNARKs, we extend the streaming paradigm of [Block et al., TCC'20]. We provide a definitional framework for elastic polynomial interactive oracle proofs for R1CS instances and design a compiler which transforms an elastic PIOP into a preprocessing argument system that supports streaming or random access to its inputs. Depending on the configuration, the prover will choose different trade-offs for time (either linear, or quasilinear) and memory (either linear, or logarithmic). We prove the existence of elastic SNARKS by presenting Gemini, a novel FFT-free preprocessing argument. We prove its security and develop a proof-of-concept implementation in Rust based on the arkworks framework. We provide benchmarks for large R1CS instances of tens of billions of gates on a single machine.

Keywords: succinct non-interactive arguments; interactive oracle proofs

1 Introduction

Succinct non-interactive arguments of knowledge (SNARKs) allow for efficient verification of NP statements. They are an essential component for a number of protocols, including private transactions [BCG+14; Zcash], verifiable computation [BCTV14; BCG+18; GGPR13], and anonymous credentials [BCC+09; GGM14]. Recent years have seen a surge of interest in SNARKs, and after much dedicated research reducing communication complexity and verifier complexity [PGHR13; BCC+16; Tha13], the cost of running the prover algorithm has emerged as the most relevant bottleneck.

Today, the most important factors are the time and memory required to run the prover algorithm. For example, in *zk-rollups*[4], some nodes in the network (called *relayers*) must produce a SNARK proving validity of a large amount of transactions. Here, the instance being proven may contain billions of constraints. Even more resource-intensive is the Filecoin network: daily, Filecoin generates proofs for about 930B constraints[5]. In both cases, the ability to prove massive statements efficiently is critical, yet many SNARK implementations today can't prove circuits of tens of billions of constraints. This work investigates time- and space-efficient SNARKSs, and the possible compromises that can be made within the same proving algorithm to get the best of both worlds.

Fast prover. A colossal effort has been put into reducing computational overheads for the prover. A long line of works [Set20; BCG20; BCG+17; BCG+18; XZZ+19] focusing on prover efficiency has led to SNARKs whose prover algorithm runs in linear (or *almost-linear*) time with respect to the instance and the witness. Some works [ZWZ+21; JW17] even rely on specialized hardware to accelerate prover computation. Unfortunately, most linear-time provers [Tha13] exploit dynamic programming techniques and as a consequence also require random access to both instance and witness, and demand space linear in the instance size. When proving large instances, this makes them prohibitively greedy in terms of memory.

Slim prover. A recent research direction [HR18; BHR+20; BHR+21] investigated, from a theoretical perspective, the possibility of a space-efficient prover. This line of work considers provers that have *streaming access* to the inputs (the statement and the witness), rather than random access to them. It has been shown [BHR+20] that a space-efficient prover needs only logarithmic memory space throughout their entire execution, but requires quasilinear computation time.

Serving diverse computing environments. Each of the above lines of research aims to optimise for a single type of complexity measure. Works which optimise for prover time assume that large amounts of memory are available, which is unlikely to be the case for considerably large instances. On the other hand, for works which focus on optimising for prover space, time efficiency is not a priority, and time overheads may be problematic in practice. In the best case, one could envisage a SNARK which simultaneously runs in linear time and uses only logarithmic memory space by accessing its inputs via streams. Unfortunately, constructing such SNARKs remains a challenging open problem. Complexity-preserving SNARKs [BC12; BCCT13], which aim to preserve both time and space complexity, are a step in the right direction. Sadly, the notion of complexity-preservation in this works allows polylogarithmic blow-ups in time and space, and this looseness makes complexity preserving SNARKs inefficient in practice. In this work, our goal is to meaningfully relax the goal of complexity preservation,

[4] `https://ethereum.org/en/developers/docs/scaling/layer-2-rollups/`
[5] `https://research.protocol.ai/sites/snarks/`

in a way that allows us to serve many different computing environments in a concretely-efficient manner.

1.1 Our results

(i) Elastic SNARKs. Roughly speaking, we consider SNARKs whose prover admits two different implementations:
  - the *time-efficient* prover $\mathcal{P}_t$, which receives as input instance and witness;
  - the *space-efficient* prover $\mathcal{P}_s$, which has streaming access to the same inputs.

Provided with this "dual-mode" framework, an elastic prover can choose which implementation to use, and allocate resources depending on the execution environment and the instance size. In addition, the two algorithms are compatible in such a way that during the execution of the protocol the space-efficient prover can stop and transcribe a (compressed) prover state. Then, the prover can switch to the time-efficient implementation, enjoying the benefits of a fast prover.

To achieve the above, we extend the notion of streams of Block et al. [BHR+20]: we study stream composition, and provide a definitional framework for streaming holographic polynomial IOPs for Rank-1 Constraint Systems (R1CSs) instances. We construct a compiler that transforms an elastic PIOP into a preprocessing argument using elastic polynomial commitment schemes.

(ii) An elastic SNARK for R1CS. We realize the above notion by constructing a novel argument system for R1CS, whose prover admits a time-efficient mode and a space-efficient mode. The two modes are compatible in such a way that it is possible to migrate state from one to the other, and produce the same final proof independently of the prover configuration.

Definition 1. *The R1CS problem asks: given a finite field $\mathbb{F}$, coefficient matrices $A, B, C \in \mathbb{F}^{N\times N}$ each containing at most $M = \Omega(N)$ non-zero entries,*[6] *and an instance vector $\mathbf{x}$ over $\mathbb{F}$, is there a witness vector $\mathbf{w}$ such that $\mathbf{z} := (\mathbf{x}, \mathbf{w}) \in \mathbb{F}^N$ and $A\mathbf{z} \circ B\mathbf{z} = C\mathbf{z}$?*

Above, "$\circ$" denotes the entry-wise product. We use standard Landau notation. When referring to time efficiency, the asymptotic number of cryptographic operations (that is, group operations) will be denoted by O_λ, to distinguish them from (less expensive) field operations, that instead we denote with standard big-O notation. Our main contribution is the following theorem:

Theorem 1 (informal). *There exists an elastic SNARK for $\mathcal{R}_{\mathrm{R1CS}}$ whose prover admits two implementations:*

[6] Note that $M = \Omega(N)$ without loss of generality because if $M < N/3$ then there are variables of $\mathbf{z}$ that do not participate in any constraint, which can be dropped. Thus the main size measure for R1CS is the sparsity parameter M.

– *the* time-efficient *prover runs in $O_\lambda(M)$ time and $O(M)$ space;*
– *the* space-efficient *prover runs in $O_\lambda(M \log^2 M)$ time and $O(\log M)$ space,*

where M is the number of non-zero entries in the R1CS instance. The proof can be verified in $O_\lambda(|\mathbf{x}| + \log M)$ time, and has size $O(\log M)$.

To achieve the above, we study the commitment of Kate et al. [KZG10] from the perspective of an elastic commitment scheme, which involves constructing a streaming interface for commitment and opening algorithms. Then, we construct an elastic scalar product protocol that runs in linear-time and linear-space, or quasi-linear time and log-space. Our scalar product argument is based on the sumcheck protocol which, thanks to its recursive nature, allows us to easily migrate from a space-efficient instance to a time-efficient one. Using the above elastic scalar product protocol, we build a polynomial IOP [BCS16] for R1CS.
Finally, we give a compiler which uses elastic polynomial commitment schemes and elastic polynomial IOPs to construct elastic cryptographic arguments. This modularity is beneficial not only for protocol design but also reflects the actual implementation. On the one hand, when studying a complex protocol, one can still isolate the cryptographic components from the information-theoretic part to study its complexity and its security. On the other hand, the implementation can benefit from an abstraction layer that reduces the implementation overhead.
Using similar techniques, we provide a preprocessing SNARK with the same complexity.

(iii) Implementation. We implement the construction of Theorem 1 in Rust using the arkworks ecosystem [ark]. Our implementation consists of the main preprocessing argument, and a *non-preprocessing argument* (where the verification procedure is assumed access to the R1CS instance in full). Extending the library with streaming-friendly primitives required a notable engineering effort that we believe could be of independent interest for future space-efficient projects. In Section 2.7, we give an overview of the relevant design choices and provide a number of algorithmic optimizations.

(iv) Evaluation. While there are plenty of benchmarks publicly available for time-efficient SNARKs, few works evaluate SNARKs on large circuits. To the best of our knowledge, the largest instance size ever proven in the literature is DIZK [WZC+18], with a maximal instance of size 2^{31}, and using a cluster of 20 machines in 256 executors.
Our benchmarks, described more in-depth in Section 2.8 show the following:
– Gemini is able to prove instances of arbitrary size. In particular, using a single machine with around 1 GB of memory budget, we are able to run benchmarks with instances of 2^{32} for the preprocessing argument. If the verifier is allowed to read the entire circuit (that is, a non-preprocessing argument), we were able to carry out proofs for 2^{35} constraints. In contrast, the largest instance ever proven in the literature [WZC+18] is only 2^{31}.

– Gemini is concretely and economically efficient. The preprocessing protocol can prove instances of size 2^{31} within two days and save about 82% (about 400 USD) of expenses when comparing with DIZK on Amazon EC2.
– Gemini provides succinct proofs and verifiers. For instances of size 2^{35}, the proof size is about 27 KB and the verification time is below 30 ms.

1.2 Related work

There is a long line of work on improving the time complexity of SNARK provers, both asymptotically and concretely; this has culminated in SNARKs with linear-time provers. See [GLS+21] and references therein. However, these optimizations typically come at the expense of space complexity, which is typically linear in the computation size either due to the use of FFTs or dynamic programming algorithms.

Simultaneously optimizing for time and space efficiency was first considered for succinct arguments in [BC12], via the notion of *complexity preservation.* This roughly means that the time and space to prove a computation must be asymptotically close to those for merely running the computation itself. Further constructions of complexity preserving succinct arguments were given in [BCCT13; HR18; BHR+20; BHR+21]. In all of these constructions space efficiency is achieved at the expense of a somewhat higher time complexity, due to the need for either a non-black-box use of cryptography or the need to perform multiple (indeed, logarithmically many) passes on the computation transcript.

Our goal in this work is to study succinct arguments that offer multiple algorithms for the same prover that optimize for different settings, e.g., for time efficiency or space efficiency. Moreover, prior works that study streaming SNARK provers were theoretical, while in this work we additionally study streaming implementations with concrete efficiency.

2 Techniques

In Section 2.1, we outline the streaming model. After setting some terminology, we state our main theorem, which is based on elastic polynomial commitment schemes and elastic probabilistic proofs. We describe then the two components separately: first, we describe an elastic polynomial commitment based on KZG in Section 2.3, and then we construct a polynomial PIOP for R1CS, which is itself based on a novel elastic scalar-product argument.

2.1 Elasticity and a streaming model

The notion of elasticity refers to having multiple realizations of the same algorithm (more precisely, function) for use in different situations. Specifically in this work:

Elasticity means that we aim for two realizations: a **time-efficient realization** for a setting where time complexity is most important, possibly at the expense of space complexity; and a **space-efficient realization** for a setting where space complexity (i.e., memory consumption) is most important, possibly at the expense of time complexity.

This means that in theorem statements, and in their proofs, we will consider two realizations with different complexities for the same functionality (e.g., the SNARK prover algorithm).

Time-efficient algorithms are a familiar concept. To discuss space-efficiency, however, we must consider *streaming algorithms*, which receive their inputs in *streams* (small pieces at a time) so that we can design algorithms that use less memory than the size of their inputs. Below we describe: (i) a formal model of streams, and (ii) a notion of streaming algorithms, and how their efficiency behaves under composition.

Streams and streaming oracles. A *stream* is a tuple consisting of an alphabet Σ, a well-ordered countable set I, and a sequence $K \in \Sigma^I$. Streams can be accessed via special oracles: if K is a sequence, the *streaming oracle* $\mathcal{S}(K)$ of K takes two input commands, start and next; the oracle simply responds to the i-th next command with the i-th element of K; the stream $\mathcal{S}(K)$ can be reset to the first element in the sequence using the start command, in case earlier elements of the stream need to be viewed again. However, the streaming oracle does not allow random access to elements of K. In the full version, we define streaming relations in which the instance and the witness are given as streams.

Streaming algorithms. A *streaming algorithm* is an algorithm that has access to all of its inputs via streaming oracles and produces a stream as its output, by yielding the next element on upon receiving the next command. The complexity of a streaming algorithm is measured in terms of its time complexity, space complexity, and the number of passes that it makes over each input stream.

Any binary operation over an alphabet can be viewed as a streaming algorithm which takes as input two sequences K and K' over the same alphabet Σ that are indexed by the same set I. In this case, the binary operation acts on successive pairs of elements of K and K', to produce a new stream on the fly. For instance, let $\mathbf{f}, \mathbf{g}$ be two vectors over a prime field $\mathbb{F}$ of order p, and $\mathcal{S}(\mathbf{f})$, $\mathcal{S}(\mathbf{g})$ (respectively) their canonical streams.[7] The stream $\mathcal{S}(\mathbf{f} + \rho\mathbf{g})$ for two vectors $\mathbf{f}, \mathbf{g}$ over a field $\mathbb{F}$ and scalar $\rho \in \mathbb{F}$ can be evaluated as a new stream using $\mathcal{S}(\mathbf{f})$ and $\mathcal{S}(\mathbf{g})$, by responding to each next query in the following way: first query $\mathcal{S}(\mathbf{f})$ to obtain the i-th entry f_i of $\mathbf{f}$, then query $\mathcal{S}(\mathbf{g})$ to obtain g_i, and finally respond with $f_i + \rho g_i$.

Since a streaming algorithm produces a stream as output, multiple streaming algorithms can be *composed* so that the output stream produced by one algorithm acts as the input stream for the next algorithm. The time and space complexity and number of input passes of streaming algorithms behave predictably under composition. If $\mathcal{A}$ is a streaming algorithm with time complexity $t_{\mathcal{A}}$, space complexity $s_{\mathcal{A}}$, and $k_{\mathcal{A}}$ input passes, and $\mathcal{B}$ is a streaming algorithm with time complexity $t_{\mathcal{B}}$, space complexity $s_{\mathcal{B}}$, and $k_{\mathcal{B}}$ input passes, then $\mathcal{A}$ composed with $\mathcal{B}$ has time complexity $t_{\mathcal{A}} + k_{\mathcal{A}} t_{\mathcal{B}}$, space complexity $s_{\mathcal{A}} + s_{\mathcal{B}}$, and $k_{\mathcal{A}} k_{\mathcal{B}}$ input passes.

[7] The canonical stream of a vector consists of the sequence of its entries, from last to first.

2.2 A modular construction of elastic SNARKs

Many succinct arguments are built in two steps. First, construct an information-theoretic probabilistic proof in a model where the verifier has a certain type of query access to the prover's messages. Second, compile the probabilistic proof into an interactive succinct argument, via a cryptographic commitment scheme that "supports" this query access.[8] Finally, if non-interactivity is desired, apply the Fiat–Shamir transformation [FS86]. This modular approach has enabled researchers to study the efficiency and security of simpler components, which has facilitated much progress in succinct arguments.

We observe that the techniques used in [CHM+20; BFS20] to build a compiler from IOPs to preprocessing arguments *preserve elasticity*: if the ingredients to the approach are elastic then the resulting SNARK is elastic. In more detail, the compiler involves two ingredients.

– *Polynomial IOPs.* A probabilistic proof in which the prover sends polynomial oracles to the verifier, who accesses them via polynomial evaluation queries. This is an interactive oracle proof [BCS16; RRR16] where query access to prover messages is changed from "point queries" to "polynomial evaluation queries".
– *Polynomial commitments.* A cryptographic primitive that enables a sender to commit to a polynomial $\mathbf{f} \in \mathbb{F}[X]$ of bounded degree, and later prove that $\mathbf{f}(z) = v$ for given $v, z \in \mathbb{F}$.

If the polynomial IOP is additionally *holographic* then the resulting succinct argument is a *preprocessing argument*, which means that it is possible, in an offline phase, to perform a public computation that enables sub-linear verification later on. The lemma below summarizes how elasticity is preserved. The formal statement (and its proof) are relative to the formalism for streaming algorithms that we outlined in Section 2.1.

Theorem 2 (informal)**.** *Suppose that we are given the following ingredients.*

– *A public-coin polynomial IOP for a relation $\mathcal{R}$ with: (i) time-efficient prover time $t_{\mathcal{P}}$; (ii) space-efficient prover space $s_{\mathcal{P}}$ with $k_{\mathcal{P}}$ passes; (iii) s oracles; (iv) query complexity q (v) verifier complexity $t_{\mathcal{V}}$*
– *A polynomial commitment scheme PC with (i) time-efficient commit and open time $t_{\mathsf{PC.Com}}$; (ii) space-efficient commit (and open) space $s_{\mathsf{PC.Com}}$ with $k_{\mathsf{PC.Com}}$ passes; and (iii) checking time $t_{\mathsf{PC.Check}}$.*

Then there exists an interactive argument system for the relation $\mathcal{R}$ with (i) time-efficient prover time $t_{\mathcal{P}} + \mathsf{s} \cdot t_{\mathsf{PC.Com}} + q \cdot t_{\mathsf{PC.Com}}$; (ii) space-efficient prover space $s_{\mathcal{P}} + \mathsf{s} \cdot s_{\mathsf{PC.Com}}$ with $q \cdot k_{\mathsf{PC.Com}} \cdot k_{\mathcal{P}}$ passes; and (iii) verifier complexity $t_{\mathcal{V}} + q \cdot t_{\mathsf{PC.Check}}$. Moreover, the argument system is preprocessing if the given polynomial IOP is holographic (with time and space properties similarly preserved by the transformation).

[8] The argument prover and argument verifier emulate the underlying probabilistic proof, with the argument prover sending commitments to proof messages and sending answers to queries together with commitment openings to authenticate those answers.

Roughly speaking, the argument prover commits to each polynomial oracle via the polynomial commitment scheme, and answers polynomial evaluation queries by sending the evaluation along with a proof that it is consistent with the corresponding polynomial commitment. The security and most efficiency measures are studied in [CHM+20; BFS20]. Less obvious is how space complexity is affected.

A streaming implementation of the PIOP prover does not necessarily produce all of its output polynomial streams one by one, and therefore the space complexity of the resulting argument prover is not, e.g., just the sum $s_{\mathcal{P}} + s_{\mathsf{PC.Com}}$ of the PIOP prover space and the PC commitment algorithm space. Indeed, if the PIOP prover's message polynomials all depend on the same input stream, it might be advantageous to produce two polynomials at the same time to avoid making extra passes over the input stream.[9] Furthermore, the commitment algorithm may require several passes over a single input polynomial, so that the argument prover must run the PIOP prover several times in order to completely commit to each polynomial, keeping partially computed commitments to each polynomial in memory. Such considerations lead to the space-efficient argument prover having space complexity $s_{\mathcal{P}} + \mathsf{s} \cdot s_{\mathsf{PC.Com}}$ with $q \cdot k_{\mathsf{PC.Com}} \cdot k_{\mathcal{P}}$ passes. Our PIOP construction actually satisfies the strong property that each polynomial can be produced independently without rerunning the entire prover algorithm, which reduces the space complexity to $s_{\mathcal{P}} + s_{\mathsf{PC.Com}}$.

Remark 1 (types of polynomials). The above discussion is deliberately ambiguous about certain aspects: are the polynomials univariate or multivariate? are the polynomials represented as vectors of coefficients or as vectors of evaluations (or vectors in some other basis)? These details do not matter for Theorem 2 as long as the two components "match up": if the PIOP outputs polynomials represented in a way that is compatible with how the PC scheme expects inputs. Nevertheless, in this paper we focus on the case of univariate polynomials represented as vectors of coefficients, because our construction and implementation are in this setting.

Remark 2 (multilinear vs. univariate). The fact that the approach in [CHM+20; BFS20] preserves space efficiency in the case of multilinear polynomials represented over the boolean hypercube was used in [BHR+20; BHR+21]. Theorem 2 is a straightforward observation about [CHM+20; BFS20] that additionally considers elasticity. In particular, we believe that the constructions in [BHR+20; BHR+21] could be shown to have elastic realizations, by showing that the underlying multilinear PIOP and multilinear PC schemes have elastic realizations. We choose to work with univariate polynomials, instead of multilinear polynomials, because they have seen more success in real world deployments, and thus focus our investigation on the concrete efficiency of elastic SNARKs based on univariate polynomials. We leave the study of concrete efficiency of elastic SNARKs based on multilinear polynomials to future work.

[9] For example, if one polynomial consists of all of the even coefficients of another, one can produce streams of the coefficients of both polynomials simultaneously, in half the number of passes required to compute streams of each polynomial one at a time.

Remark 3 (elastic setup and indexer). For any succinct argument, elasticity is a desirable property as the size of the statement to be proven increases. In particular, we are going to focus on elasticity of the prover, which is the current bottleneck for proving large instances.

– *Setup.* We assume the existence of a setup algorithm that samples the public parameters of the system. Despite its complexity can be linear (or more!) in the statement size, we do not discuss setup algorithms in this paper for two reasons: (i) known setup algorithms have straightforward realizations that are simultaneously efficient in time and space (that is, there is less of a tension between optimizing for time or for space as there is for the prover); (ii) public parameters are typically sampled via "cryptographic ceremonies" that realize the setup functionality via secure multi-party protocols [BGM17], and so it is more relevant to discuss the time and space efficiency of these ceremonies.
– *Indexer.* In the case of preprocessing arguments, there is an indexer algorithm that produces the so-called *proving key* and *verification key*. The indexer in our construction and implementation is elastic, but we will not focus on it since all ideas relevant for the indexer can be inferred from the proving algorithm.

2.3 An elastic realization of the KZG polynomial commitment scheme

We use a univariate polynomial commitment scheme from [KZG10] to construct our SNARK (see Section 2.2). Below we review this scheme and explain how to realize it elastically.

Review: a polynomial commitment from [KZG10]. The setup algorithm samples and outputs public parameters for the scheme to support polynomials of degree at most $D \in \mathbb{N}$: the description of a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, G, H, e)$;[10] the commitment key $\mathsf{ck} := (G, \tau G, \dots, \tau^D G) \in \mathbb{G}_1^{D+1}$ for a random field element $\tau \in \mathbb{F}_q$; and the receiver key $\mathsf{rk} := (G, H, \tau H) \in \mathbb{G}_1 \times \mathbb{G}_2^2$. The commitment to a polynomial $\mathbf{p} \in \mathbb{F}_q[X]$ of degree at most D is computed as $C := \langle \mathbf{p}, \mathsf{ck} \rangle = \mathbf{p}(\tau)G \in \mathbb{G}_1$. Subsequently, to prove that the committed polynomial $\mathbf{p}$ evaluates to v at $z \in \mathbb{F}_q$, the committer computes the witness polynomial $\mathbf{w}(X) := (\mathbf{p}(X) - \mathbf{p}(z))/(X - z)$, and outputs the evaluation proof $\pi := \langle \mathbf{w}, \mathsf{ck} \rangle = \mathbf{w}(\tau)G \in \mathbb{G}_1$. Finally, to verify the evaluation proof, the receiver checks that $e(C - vG, H) = e(\pi, \tau H - zH)$.

Elastic realization. An elastic realization of the above scheme requires a time-efficient realization and a space-efficient realization for each relevant algorithm of the scheme. Here we do not discuss the setup algorithm, as it has a natural time-and-space-efficient realization (cf. Remark 3). We do not discuss the verification algorithm either, because it only involves a constant number of scalar multiplications and pairings. Our focus is thus on the commitment and opening algorithm.

– *Commitment algorithm.* We are given streams of the commitment key elements $\{\tau^i G\}_{i=0}^d$ and of the coefficients $\{p_i\}_{i=0}^d$ of the polynomial $\mathbf{p}(X) = \sum_{i=0}^d p_i X^i$

[10] Here $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T| = q$, G generates $\mathbb{G}_1$, H generates $\mathbb{G}_2$, and $e \colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a non-degenerate bilinear map.

to be committed. We compute the commitment $C = \sum_{i=0}^{d} p_i \tau^i G$ by multiplying each coefficient-key pair $(p_i, \tau^i G)$ together and adding them to a running total. Each scalar-multiplication of $p_i \cdot \tau^i G$ is performed via a double-and-add algorithm in constant space and linear time.
The above operation is also known as multi-scalar multiplication or multi-exponentiation in the literature, and there exists different algorithms, such as Pippenger's [Pip80], that can greatly improve the concrete efficiency of the multi-scalar multiplication by reducing the number of group operations. Unfortunately, the algorithm requires random access to the vectors of scalars $(\mathbf{p})$ and bases (ck) and, despite Pippenger itself can be transformed into a streaming algorithm (trading off constant memory), the best performance in practice is achieved by setting a constant buffer and computing the result $C = \langle \mathbf{p}, \mathsf{ck} \rangle$ via multiple executions of Pippenger's algorithm. We investigate more non-naïve streaming algorithms in Section 2.7.
– *Opening algorithm.* We are given the same streams as above, and an opening location z. By rearranging the expression for the witness polynomial $\mathbf{w}(X) = (\mathbf{p}(X) - \mathbf{p}(z))/(X - z)$, we can stream the coefficients $\{w_i\}_{i=0}^{d-1}$ of $\mathbf{w}(X)$ via Ruffini's rule: $w_i \coloneqq p_{i+1} + w_{i+1} z$. The evaluation proof $\pi = \sum_{i=0}^{d-1} w_i \tau^i G$ is computed in the same way as the commitment algorithm.

Note that the recurrence relation in the opening algorithm uses w_{j+1} to compute w_j, which means that $\mathbf{w}(X)$ is computed from its highest-order coefficient to its lowest. In turn, this means that the commitment key ck and the polynomial $\mathbf{p}(X)$ are streamed from highest-degree to lowest-degree coefficient. Setup and commitment algorithms are agnostic to the order elements are being streamed. The above discussion implies the following (informal) lemma.

Lemma 1 (informal). *The polynomial commitment scheme of [KZG10] has an elastic realization.*

2.4 An elastic scalar-product protocol

A scalar-product protocol enables the prover to convince the verifier that the scalar product of two committed vectors equals a certain target value. In the literature, scalar-product protococols [BCG20] are also known as inner-product arguments or IPAs [BCC+16; DRZ20; BMM+21]. Many constructions of succinct arguments internally rely on scalar-product protocols as their main component for proving NP statements [BCC+16; PLS19; BCG20]. The PIOP for R1CS that we construct in Sections 2.5 and 2.6 relies on a PIOP for scalar products where the prover has two realizations: (i) one that runs in linear-time and linear-space; and (ii) one that runs in quasilinear-time and logarithmic-space.

Definition 2. *A* **PIOP for scalar products** *is a PIOP where the verifier receives as input $(\mathbb{F}, N, u)$ and has (polynomial evaluation) query access to $\mathbf{f}, \mathbf{g} \in \mathbb{F}^N$, and checks with the help of the prover that $\langle \mathbf{f}, \mathbf{g} \rangle = u$.*

Theorem 1 (informal). *There is a PIOP for scalar products with proof length $O(N)$, query complexity $O(\log N)$, verifier time $O(\log N)$, and prover that has two realizations:*

– *a time-efficient realization that runs in time $O(N)$ and space $O(N)$;*
– *a space-efficient realization that runs in time $O(N \log N)$ and space $O(\log N)$ in $O(\log N)$ passes.*

In the remainder of this section, we outline the scalar-product protocol, deferring to the full version formal security proofs and a more in-depth discussion of the protocol. We also note that our construction uses two slightly different protocols, one for *twisted scalar-products* [BCG20], where $\langle \mathbf{f} \circ \mathbf{y}, \mathbf{g} \rangle = u$ for a vector $\mathbf{y}$ of the form $\mathbf{y} = (1, \rho_0) \otimes (1, \rho_1) \otimes \cdots \otimes (1, \rho_{n-1})$ where $n \coloneqq \log N$ (we recall that by log we denote the ceil of the logarithm base 2), and one for verifying Hadamard product relations $\mathbf{f} \circ \mathbf{g} = \mathbf{h}$. These only require small modifications to the scalar-product protocol.

We proceed in three steps. First, in Section 2.4.1 we describe how to reduce checking a scalar product to checking tensor products of univariate polynomials. Then, we describe a tensor product protocol in Section 2.4.2. Finally, in Section 2.4.3 we describe how to realize this protocol in an elastic way.

2.4.1 Verifying scalar products using the sumcheck protocol

Consider two vectors $\mathbf{f}, \mathbf{g} \in \mathbb{F}^N$ with $\langle \mathbf{f}, \mathbf{g} \rangle = u$ as in Definition 2. The verifier has polynomial evaluation query access to polynomial evaluations of $\mathbf{f}$ and $\mathbf{g}$. That is, the verifier can obtain evaluations of the polynomials $\mathbf{f}(X) = \sum_{i=0}^{N-1} f_i X^i$ and $\mathbf{g}(X) = \sum_{i=0}^{N-1} g_i X^i$ at any point $x \in \mathbb{F}$. Note that the product polynomial $\mathbf{h}(X) \coloneqq \mathbf{f}(X) \cdot \mathbf{g}(X^{-1})$ has $\langle \mathbf{f}, \mathbf{g} \rangle = \sum_{i=0}^{N-1} f_i g_i$ as the coefficient of X^0, because for every $i, j \in [N]$ the powers of X associated with f_i and g_j multiply together to give X^0 if and only if $i = j$. Therefore, to check the scalar-product $\langle \mathbf{f}, \mathbf{g} \rangle = u$, it suffices to check that the coefficient of X^0 in the product polynomial $\mathbf{h}(X)$ equals u.

However, this check must somehow be performed *without the prover actually computing* $\mathbf{h}(X)$. This is because the fastest algorithm for computing $\mathbf{h}(X)$ requires $O(N \log N)$ time and $O(N)$ space (via FFTs), which is neither time-efficient nor space-efficient. On the other hand, the scalar product $\langle \mathbf{f}, \mathbf{g} \rangle = u$ can be checked (directly) in time $O(N)$ and space $O(1)$, which leaves open the possibility of a scalar-product protocol where the prover does better than computing $\mathbf{h}(X)$ explicitly (and then running some protocol).

This issue is addressed in prior work if the verifier can query the *multilinear* polynomials $\widehat{\mathbf{f}}(\mathbf{X})$ and $\widehat{\mathbf{g}}(\mathbf{X})$ associated to the vectors $\mathbf{f}, \mathbf{g} \in \mathbb{F}^N$: we index the entries of $\mathbf{f}$ using binary vectors, and $f_i = f_{b_0, \ldots, b_{n-1}}$ is the coefficient of $X_0^{b_0} \cdots X_{n-1}^{b_{n-1}}$, where $(b_0, \ldots, b_n)$ is the binary decomposition of i. From previous results [Tha13; XZZ+19; BCG20], we have the following lemma.

Lemma 2. *Let $\mathbb{F}$ be a finite field and N be a positive integer. Define $n = \log N$; the sumcheck protocol for*

$$\frac{1}{2^n} \sum_{\boldsymbol{\omega} \in \mathcal{H}^n} (\widehat{\mathbf{f}} \cdot \widehat{\mathbf{g}})(\boldsymbol{\omega}) = u \ . \tag{1}$$

for $\mathcal{H} = \{-1, 1\}$ *and two multilinear polynomials* $\widehat{\mathbf{f}}(X_0, \ldots, X_{n-1})$ *and* $\widehat{\mathbf{g}}(X_0, \ldots, X_{n-1})$ *has the following properties: soundness error is* $O(\log N/|\mathbb{F}|)$ *(as a reduction to claims about polynomial evaluations); round complexity is* $O(\log N)$*; the prover uses* $O(\log N)$ *field operations; and the verifier uses* $O(\log N)$ *field operations.*

Then, one can use the (multivariate) sumcheck protocol of [LFKN92] to reduce $\langle \mathbf{f}, \mathbf{g} \rangle = u$ to two evaluation queries $\widehat{\mathbf{f}}(\boldsymbol{\rho})$ and $\widehat{\mathbf{g}}(\boldsymbol{\rho})$, where $\boldsymbol{\rho} := (\rho_0, \ldots, \rho_{n-1}) \in \mathbb{F}^n$ are the random verifier challenges used in the sumcheck protocol. Crucially, the prover algorithm in the sumcheck protocol applied to the two product of two multilinear polynomials also has a space-efficient realisation which runs in time $O(N \log N)$ and space $O(\log N)$ [CMT12], which would provide an elastic solution in this multilinear regime.

In our setting the verifier can only query the *univariate* polynomials $\mathbf{f}(X)$ and $\mathbf{g}(X)$ associated with the vectors $\mathbf{f}, \mathbf{g} \in \mathbb{F}^N$. Nevertheless, we follow a similar approach, by running the sumcheck protocol on the *multivariate* polynomials $\widehat{\mathbf{f}}(\mathbf{X})$ and $\widehat{\mathbf{g}}(\mathbf{X})$, producing two claimed evaluations $\widehat{\mathbf{f}}(\boldsymbol{\rho}) = u$ and $\widehat{\mathbf{g}}(\boldsymbol{\rho}) = u'$. We check that these claimed evaluations are consistent with $\mathbf{f}$ and $\mathbf{g}$ using evaluations of the *univariate* polynomials $\mathbf{f}(X)$ and $\mathbf{g}(X)$.

Remark 4 (unstructured fields). While other probabilistic proofs using univariate polynomials, such as the low-degree test in [BBHR18], require the size of the field $\mathbb{F}$ to be *smooth*, so that the field contains high-degree roots of unity or structured linear subspaces. In contrast, using the strategy above, our scalar-product protocol works with univariate polynomials over any sufficiently large field. This allows a much wider range of parameter choices for the [KZG10] polynomial commitment scheme which is likely to lead to better concrete efficiency.

2.4.2 A tensor-product protocol

Consider the claimed evaluation $\widehat{\mathbf{f}}(\boldsymbol{\rho}) = v$. To check that this is correct using univariate polynomial evaluations, note that $\widehat{\mathbf{f}}(\mathbf{X})$ and $\mathbf{f}(X)$ have the same coefficients, and moreover partially evaluating $\widehat{\mathbf{f}}(\mathbf{X})$ by setting X_0 equal to ρ_0, the polynomial $\widehat{\mathbf{f}}(\rho_0, X_1, \ldots, X_{\log N-1})$ has the same coefficients as the polynomial $\mathbf{f}'(X) := \mathbf{f}_e(X) + \rho_0 \cdot \mathbf{f}_o(X)$. Here, $\mathbf{f}_e(X)$ and $\mathbf{f}_o(X)$ are the unique odd and even parts of $\mathbf{f}(X)$, defined by $\mathbf{f}(X) = \mathbf{f}_e(X^2) + X\mathbf{f}_o(X^2)$.

This suggests a protocol where the prover sends $\mathbf{f}'(X)$ to the verifier. If the verifier can check that $\mathbf{f}'(X)$ was correctly computed from $\mathbf{f}(X)$, then the original problem of consistency between $\mathbf{f}(X)$ and an evaluation of $\widehat{\mathbf{f}}(X_0, \ldots, X_{\log N-1})$ is reduced to checking consistency between $\mathbf{f}'(X)$ and an evaluation of $\widehat{\mathbf{f}}(\rho_0, X_1, \ldots, X_{\log N-1})$. Repeating this reduction with every value ρ_j, the prover and verifier eventually arrive at a claim about constant-degree polynomials, which the prover can send to the verifier allowing the verifier to check autonomously.

To check that $\mathbf{f}'(X)$ is consistent with $\mathbf{f}(X)$, the verifier can sample a random challenge point $\beta \in \mathbb{F}^\times$ (where $\mathbb{F}^\times$ denotes the multiplicative group of $\mathbb{F}$), and make polynomial evaluation queries in order to check the following equations:

$$\mathbf{f}'(\beta^2) = \mathbf{f}_e(\beta) + \rho_0 \cdot \mathbf{f}_o(\beta) = \frac{\mathbf{f}(\beta) + \mathbf{f}(-\beta)}{2} + \rho_0 \cdot \frac{\mathbf{f}(\beta) - \mathbf{f}(-\beta)}{2\beta} \ . \tag{2}$$

This is reminiscent of a similar reduction in [BBHR18] used to construct a low-degree test for univariate polynomials. By the Schwartz–Zippel lemma, the check passes with small probability unless $\mathbf{f}'(X)$ was computed correctly.

Noting that $\widehat{\mathbf{f}}(\boldsymbol{\rho}) = \langle \mathbf{f}, \otimes_{j=0}^{n-1}(1, \rho_j) \rangle$, this procedure gives a univariate polynomial IOP for the following relation:

Definition 3. *The* **tensor-product** *relation $\mathcal{R}_{\mathrm{TC}}$ is the set of tuples*

$$(\mathbb{i}, \mathbb{x}, \mathbb{w}) = (\perp, (\mathbb{F}, N, \rho_0, \ldots, \rho_{n-1}, u), \mathbf{f})$$

where $n = \log N$, $\mathbf{f} \in \mathbb{F}^N$, $u \in \mathbb{F}$, and $\langle \mathbf{f}, \otimes_j (1, \rho_j) \rangle = u$.

We give full details of the tensor-product protocol in the full version of this paper. In fact, the tensor check will be useful not only as part of our scalar-product protocol, but also more generally as part of simple checks that take place as part of our R1CS protocols (as described in Sections 2.5 and 2.6).

2.4.3 Elastic realization of the prover algorithm

Most complexity measures claimed in Theorem 1 follow straightforwardly from the sumcheck protocol described in Lemma 2. What remains is to describe an elastic realization of the prover algorithm for the tensor-product protocol.

The prover's task is to compute the polynomials $\mathbf{f}^{(j)}$ for each round $j \in [n]$. Given $\mathbf{f}^{(j-1)}$, which has degree $O(N/2^j)$, the prover can compute $\mathbf{f}^{(j)}$ in $O(N/2^j)$ operations via Equation 2. Summing up the prover costs for $j \in [n]$ gives $O(N)$ operations. Therefore, it is easy to see that the tensor-product protocol has a linear-time prover realisation.

Next, we give a space-efficient prover realisation that uses logarithmic space.

Logarithmic space. We aim for the prover to run in logarithmic space complexity, given streaming access to $\mathbf{f}$ and $\mathbf{g}$. This is more challenging than the time-efficient case, as the prover cannot store $\mathbf{f}^{(j-1)}$ to help it compute $\mathbf{f}^{(j)}$, as this would require linear space complexity (for small j). Instead, the prover computes each $\mathbf{f}^{(j)}$ from scratch using streams of $\mathbf{f}$.

We begin by explaining how the prover can *produce* a stream of $\mathbf{f}^{(j)}$ efficiently, given streaming access to $\mathbf{f}$, in a similar way to streaming evaluations of multivariate polynomials and low-degree extensions in [BHR+21; BHR+20; CMT12]. Our main contribution here is to show that $\mathbf{f}^{(j)}$ can be evaluated in $O(N)$ time and $O(\log N)$ space, saving a logarithmic factor over prior work. Then, we explain how to perform the consistency checks.

– *Streaming* $\mathbf{f}^{(j)}$. Let $\mathbf{f} = \sum_{i=0}^{N-1} f_i X^i$. We can compute $\mathbf{f}' = \sum_{i=0}^{N/2-1}(f_{2i} + \rho f_{2i+1})X^i$ from a stream of coefficients of $\mathbf{f}$ by reading each pair of coefficients f_{2i}, f_{2i+1} from the stream, and produce the next coefficient $f'_i := f_{2i} + \rho f_{2i+1}$

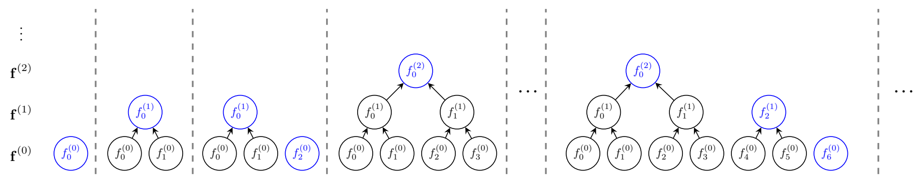

Fig. 1: A streaming algorithm for computing the coefficients of $\mathbf{f}^{(j)}$ from $\mathbf{f}^{(0)} := \mathbf{f}$. Nodes in blue denote the coefficients that are stored in memory at any moment.

of $\mathbf{f}'$. This process uses a constant amount memory space, storing f_{2i}, and f_{2i+1} and deleting them immediately after computing f'_i. Each coefficient of $\mathbf{f}'$ costs two arithmetic operations to compute.
The prover can produce the stream $\mathcal{S}(\mathbf{f}^{(j)})$ for $\mathbf{f}^{(j)}$ by applying the same idea recursively as follows. Initialise a stack Stack consisting of pairs $(j, x) \in [\log N] \times \mathbb{F}$, and a list of challenges $\rho_0, \dots, \rho_j$. To generate $\mathcal{S}(\mathbf{f}^{(j)})$, the prover

- If the top element in the stack is of the form (j, y) for some $y \in \mathbb{F}$, pop it and return y.
- If the top two elements in the stack are of the form (k', x') and (k, x) with $k = k'$ (and $k < j$), then pop them and push $(k{+}1,\ x{+}\rho_k\, x')$, where $x{+}\rho_k\, x'$ is equal to $f^{(j)}_{k+1}$ (recall that the values are streamed from last to first index);
- Otherwise, query the stream $\mathcal{S}(\mathbf{f})$ for the next element $x \in \mathbb{F}$ and add $(0, x)$ to the stack.

The stack Stack must be initialized with some zero-entries if $N \neq 2^n$ (for instance, where N is odd) for correctness, but we avoid discussing this case here for simplicity. A visual representation of this process is displayed in Figure 1. This procedure produces a stream of $\mathbf{f}^{(j)}$ from a stream of $\mathbf{f}$ in $O(N)$ and using $\log N$ memory space (since the stack Stack holds at most $\log N$ elements at any time).

– *Space-efficient tensor check.* The verifier must perform consistency checks to make sure that each polynomial $\mathbf{f}^{(j)}$ was correctly computed from $\mathbf{f}^{(j-1)}$, and similarly for $\mathbf{g}^{(j)}$. This check requires the computation of $\mathbf{f}^{(0)}, \dots, \mathbf{f}^{(n-1)}$. We compute them in parallel with a minor modification to the folding algorithm illustrated in Figure 1. Instead of returning only when the top of the stack has a particular index, we always output the top element in the stack. We thus construct a streaming algorithm $\mathcal{S}(\mathbf{f}^{(0)}, \dots, \mathbf{f}^{(n-1)})$ that returns elements of the form $(j, x) \in [n] \times \mathbb{F}$ where x is the next coefficient of the polynomial $\mathbf{f}^{(j)}$. With the above stream, it is possible to simulate the streaming oracle $\mathcal{S}(\mathbf{f}^{(j)})$ and the evaluations $\mathbf{f}^{(j)}(\beta^2)$, $\mathbf{f}^{(j)}(+\beta)$, $\mathbf{f}^{(j)}(-\beta)$, for each $j \in [n]$. In particular computing each evaluation requires storing a single $\mathbb{F}$-element; therefore, the total consistency check uses $n = \log N$ memory and N time. This allows to check Equation 2, substituting $\mathbf{f}' = \mathbf{f}^{(j)}, \mathbf{f} = \mathbf{f}^{(j-1)}$ for $j \in [n]$.

Based on the costs of maintaining the stacks for $\mathbf{f}$ and $\mathbf{g}$, and computing the coefficients of $\mathbf{q}^{(j)}$ incrementally, it follows that each round takes time $O(N)$

and space $O(\log N)$. Therefore, summing over the $O(\log N)$ rounds, the protocol requires time $O(N \log N)$ and space $O(\log N)$.

Remark 5. Based on our tensor product protocol in Section 2.4.2, one can construct a linear-time univariate sumcheck protocol with proof length $O(N)$ and query complexity $O(\log N)$, which we believe could be of independent interest for future research. There are other univariate sumcheck protocols in the literature; however, these protocols cannot be used in our setting.

The univariate sumcheck protocol in [BCR+19] is a 1-message PIOP with proof length $O(N)$ and query complexity $O(1)$. That protocol does not seem useful here, because the prover requires $O(N \log N)$ time and $O(N)$ space due to the use of FFTs. In contrast, our protocol achieves elasticity, at the cost of logarithmic round complexity and logarithmic query complexity.

Drake [Dra20] sketches a Hadamard product protocol based on univariate polynomials that does not use FFTs. That protocol, also inspired by the low-degree test in [BBHR18], may conceivably lead to a univariate sumcheck protocol that is elastic. However, no details (or implementations) of the protocol are available.

2.5 Warm-up: an elastic non-holographic PIOP for R1CS

We describe an elastic PIOP for R1CS (Definition 1), based on the elastic scalar-product protocol from Section 2.4. Due to the large verifier complexity of this protocol, we note that the verifier can also be made elastic using similar techniques to the elastic prover. We will build on this construction later in Section 2.6, and construct a *holographic* polynomial IOP with logarithmic verifier complexity.

Theorem 2 (informal). *For every finite field $\mathbb{F}$, there is a PIOP for $\mathcal{R}_{\mathrm{R1CS}}$ over $\mathbb{F}$ with the following parameters:*
– *soundness error $O(N/|\mathbb{F}|)$;*
– *round complexity $O(\log N)$;*
– *proof length $O(N)$ and query complexity $O(\log N)$;*
– *a time-efficient prover that runs in time $O(M)$ and space $O(M)$;*
– *a space-efficient prover that runs in time $O(M \log^2 N)$ and space $O(\log N)$ (with $O(\log N)$ input passes);*
– *a time-efficient verifier that runs in time $O(M)$ and space $O(M)$; and*
– *a space-efficient verifier that runs in time $O(M \log N)$ and space $O(\log N)$.*
Above, N is the matrix size and M the number of non-zero entries in an R1CS instance.

The theorem holds for *any* finite field $\mathbb{F}$, and in particular does not require any smoothness properties for $\mathbb{F}$.

To make the space-efficient realisation of the prover well-defined, we must explain how to stream an R1CS instance. Below we describe a concrete choice of streams that (i) suffices for the theorem; (ii) is realistic (as we elaborate shortly). After that we outline the polynomial IOP for R1CS .

Streaming R1CS. The R1CS problem is captured using the following indexed relation:

Definition 4. *The indexed relation $\mathcal{R}_{\text{R1CS}}$ is the set of all triples:*

$$(\mathbb{i}, \mathbb{x}, \mathbb{w}) = \big((\mathbb{F}, N, M, A, B, C), \mathbf{x}, \mathbf{w}\big)$$

where $\mathbb{F}$ is a finite field, A, B, C are matrices in $\mathbb{F}^{N\times N}$, each having at most M non-zero entries, and $\mathbf{z} \coloneqq (\mathbf{x}, \mathbf{w})$ is a vector in $\mathbb{F}^{N}$ such that $A\mathbf{z} \circ B\mathbf{z} = C\mathbf{z}$.

We define streams for each of $\mathbb{i}$, $\mathbb{x}$ and $\mathbb{w}$, with A, B and C in *sparse representation.*

Definition 1. *Let $U \in \mathbb{F}^{N\times N}$ be a matrix with M non-zero entries. The* **sparse representation** *of U consists of its coordinate-list representation. That is, the stream $\mathcal{S}(U)$ of U is the sequence of elements in the support as tuples (row, column, value), sorted by row index or column index. We denote by $\mathcal{S}_{\text{row}}(U)$ the row-major coordinate list (that is, ordering the entries of the matrix in lexicographic order with row index before column index), and by $\mathcal{S}_{\text{col}}(U)$ the column-major coordinate list.*

In our definition of streams for R1CS, we allow the *computation trace* ($A\mathbf{z}$, $B\mathbf{z}$, $C\mathbf{z}$) of an R1CS instance to be streamed as part of the witness.

Definition 2 (streaming R1CS). *The streams associated with the R1CS instance $((\mathbb{F}, N, M, A, B, C), \mathbf{x}, \mathbf{w})$ are:*
- *the* **index streams:** *streams $\mathcal{S}(A)$, $\mathcal{S}(B)$, $\mathcal{S}(C)$, the coordinate lists of the R1CS matrices, in row-major and column-major.*
- *the* **instance stream:** *stream of the instance vector $\mathcal{S}(\mathbf{x})$;*
- *the* **witness streams:** *stream of the witness $\mathcal{S}(\mathbf{w})$ and the computation trace vectors $\mathcal{S}(A\mathbf{z}), \mathcal{S}(B\mathbf{z}), \mathcal{S}(C\mathbf{z})$.*

The field description $\mathbb{F}$, instance size N, and maximum number M of non-zero entries are explicit inputs.

Including steams for $\mathcal{S}(A\mathbf{z})$, $\mathcal{S}(B\mathbf{z})$, and $\mathcal{S}(C\mathbf{z})$ makes our polynomial IOPs for R1CS space efficient even when matrix multiplication by A, B and C requires a large amount of memory and $A\mathbf{z}$, $B\mathbf{z}$ and $C\mathbf{z}$ cannot be computed element by element on the fly given streaming access to $\mathbf{x}$ and $\mathbf{w}$. On the other hand, for R1CS instances defined by many natural computations, such as a machine computation which repeatedly applies a transition function to a small state, the matrices A, B and C are *banded*; that is, their non-zero entries all lie in a thin, central diagonal band. In this case, it is easy to generate a stream of $\mathcal{S}(A\mathbf{z})$, for example, using streams $\mathcal{S}(\mathbf{x})$, $\mathcal{S}(\mathbf{w})$ and the column-major matrix stream $\mathcal{S}_{\text{col}}(A)$.

The polynomial IOP construction. We outline the PIOP construction which proves Theorem 2. The protocol adopts standard ideas from [BCR+19] and an optimization from [Gab20] for concrete efficiency. In the time-efficient realisation of our protocol, the prover takes $\mathbb{i}$, $\mathbb{x}$ and $\mathbb{w}$ as input, and the verifier receives $\mathbb{i}$ and $\mathbb{x}$. In the space-efficient realisation, these inputs are provided as streams according to Definition 2.

In the first step of the protocol, the prover sends $\mathbf{z}$ to the verifier. To check that $A\mathbf{z} \circ B\mathbf{z} = C\mathbf{z}$, the verifier replies by sending a random challenge $v \in \mathbb{F}^{\times}$

to the prover, which the prover expands into a vector $\mathbf{y}_C \coloneqq (1, \upsilon, \upsilon^2, \dots, \upsilon^{N-1})$. Then, multiplying $A\mathbf{z} \circ B\mathbf{z} = C\mathbf{z}$ on the left by $\mathbf{y}_C^\intercal$, the prover and verifier will check that

$$\langle A\mathbf{z} \circ \mathbf{y}_C,\ B\mathbf{z}\rangle = \langle C\mathbf{z}, \mathbf{y}_C\rangle \ . \tag{3}$$

The prover will send the value $u_C \coloneqq \langle C\mathbf{z}, \mathbf{y}_C\rangle \in \mathbb{F}$ to the verifier. At this point, the prover will convince the verifier that Equation 3 holds by reducing the two claims $\langle A\mathbf{z} \circ \mathbf{y}_C,\ B\mathbf{z}\rangle = u_C$ and $\langle C\mathbf{z}, \mathbf{y}_C\rangle = u_C$ to *tensor consistency checks* on $\mathbf{z}$, for which we can apply the tensor-product protocol in Section 2.4.

As a sub-protocol for the first claim, the prover and verifier run a twisted scalar product protocol, as described in Section 2.4. This generates two new claims, one about each of $A\mathbf{z}$ and $B\mathbf{z}$, leaving us with a total of three claims:

$$\begin{aligned} \langle A\mathbf{z}, \mathbf{y}_B \circ \mathbf{y}_C\rangle &= u_A \ , \\ \langle B\mathbf{z}, \mathbf{y}_B\rangle &= u_B \ , \\ \langle C\mathbf{z}, \mathbf{y}_C\rangle &= u_C \ . \end{aligned} \tag{4}$$

Here, $\mathbf{y}_B \coloneqq \otimes_j (1, \rho_j)$, where $\rho_0, \rho_1, \dots, \rho_{n-1} \in \mathbb{F}^\times$ are the random challenges sent by the verifier during the sub-protocol. Setting $\mathbf{y}_A \coloneqq \mathbf{y}_B \circ \mathbf{y}_C$, and moving the matrices A, B and C into the right input argument of the scalar-product relation, we have

$$\begin{aligned} \langle \mathbf{z}, \hat{\mathbf{a}}\rangle &= u_A \ , \\ \langle \mathbf{z}, \hat{\mathbf{b}}\rangle &= u_B \ , \\ \langle \mathbf{z}, \hat{\mathbf{c}}\rangle &= u_C \ , \end{aligned} \tag{5}$$

where $\hat{\mathbf{a}} \coloneqq \mathbf{y}_A^\intercal A$, and similarly for $\hat{\mathbf{b}}$ and $\hat{\mathbf{c}}$. Although $\mathbf{y}_B$, $\mathbf{y}_C$, and $\mathbf{y}_A$ all have a tensor structure, $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$ and $\hat{\mathbf{c}}$ will not generally have the same structure, which means that Equation 5 cannot be checked directly using the tensor-product protocol. Thus, the verifier sends another random challenge $\eta \in \mathbb{F}^\times$ to the prover. Taking linear combinations of the three claims in Equation 5 using powers of η yields a single scalar-product claim

$$\langle \mathbf{z},\ \hat{\mathbf{a}} + \eta \cdot \hat{\mathbf{b}} + \eta^2 \cdot \hat{\mathbf{c}}\rangle = u_A + \eta \cdot u_B + \eta^2 \cdot u_C \ . \tag{6}$$

The prover and verifier run a second twisted scalar-product protocol for Equation 6. This produces two new claims

$$\langle \mathbf{z}, \mathbf{y}\rangle = u_D \ , \tag{7}$$

$$\langle \hat{\mathbf{a}} + \eta \cdot \hat{\mathbf{b}} + \eta^2 \cdot \hat{\mathbf{c}}, \mathbf{y}\rangle = u_E \ , \tag{8}$$

where again, $\mathbf{y}$ is a vector with the same tensor structure as described in Section 2.4, generated using random challenges produced by the verifier.

Finally, the prover and the verifier engage in a tensor-product protocol to check Equation 7. The verifier can check Equation 8 directly, since $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$ and $\hat{\mathbf{c}}$ can be computed directly from the R1CS matrices A, B and C, along with the random challenges used throughout the R1CS protocol.

Time-efficient prover. The prover runs in linear time if the prover algorithms for the underlying scalar-product and tensor-product sub-protocols can be realized in linear time. Note that the cost of computing $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$ and $\hat{\mathbf{c}}$ is linear in the number of non-zero entries in A, B, C. As a result, the verifier also runs in linear time.

Space-efficient prover. We start by noting that the streams $\mathcal{S}(A\mathbf{z})$, $\mathcal{S}(B\mathbf{z})$ $\mathcal{S}(C\mathbf{z})$ are provided as input to the prover, and that the stream $\mathcal{S}(\mathbf{z})$ can be produced by chaining the instance stream $\mathcal{S}(\mathbf{x})$ with the witness stream $\mathcal{S}(\mathbf{w})$. In order to run the first twisted scalar-product protocol (cf. Eq. 3), the prover must compute also the stream for the tensor product vector $\mathbf{y}_C = \otimes_j(1, v^{2^j})$. This stream can be generated in linear time: during the i-th iteration, the stream of any vector of the form $\otimes_j(1, v_j)$ must return the product $\prod_{j,b_j \neq 0} v_j$, where $i = (b_0, \dots, b_{n-1})_2$ in binary. Consider the bit-string representing i: after yielding the i-th element $\prod_{j,b_j \neq 0} v_j$, the next element in the stream has index $(i-1)$, and it can be either obtained by clearing the last multiplication (that is, multiplying the previous element by v_0^{-1}) or multiplying the previous element by $v_k^{-1} v_{k-1} \cdots v_0$ (when the subtraction has a carry bit propagating k times). The stream, during the initialization phase, stores the incremental products v_0, $v_0 v_1$, $v_0 v_1 v_2$, $\dots$, $v_0 v_1 \cdots v_{n-1}$ and the "carry elements" v_0^{-1}, $v_1^{-1} v_0$, $\dots$, $v_{n-1}^{-1} v_{n-2} \cdots v_0$ Initialisation of the stream costs $O(\log N)$ field operations and $O(n)$ space; and each new element is produced with a single field multiplication. Using the same idea, it is possible to produce the streams $\mathcal{S}(\mathbf{y}_A)$ and $\mathcal{S}(\mathbf{y}_B)$.

In the second sumcheck (cf. Eq. 6), the stream for $\mathcal{S}(\hat{\mathbf{a}}) = \mathcal{S}(\mathbf{y}_A^\intercal A)$ (respectively, $\mathcal{S}(\hat{\mathbf{b}})$ and $\mathcal{S}(\hat{\mathbf{c}})$) are not implemented using trivial matrix multiplication from the row-major stream of $\mathcal{S}_{\mathrm{row}}(A)$ (resp. $\mathcal{S}_{\mathrm{row}}(B)$, $\mathcal{S}_{\mathrm{row}}(C)$) with the above tensor product. Instead of using $O(N)$ passes over the stream vector, we compute the i-th element on the fly: using the stream $\mathcal{S}_{\mathrm{row}}(A)$, all elements of i-th row $(i, j, a_{i,j})$ produced by the stream are multiplied by v^i and accumulated in the final result. Overall, producing the next element costs $O(\log N)$ multiplications.

Composing the above streams with the space-efficient realisations of the scalar product and tensor-product sub-protocols described in Section 2.4, we obtain a space-efficient prover algorithm which runs in quasilinear time and logarithmic space: overall, the non-holographic protocol can be run in $O(M \log^2 N)$ time and $O(\log N)$ space.

2.6 Elastic holographic PIOP for R1CS

The verifier complexity in the non-holographic protocol for R1CS described in Section 2.5 is linear in the size of the R1CS instance. This is because in order to run a scalar-product protocol to check Equation 6, the verifier must compute $\hat{\mathbf{a}}, \hat{\mathbf{b}}, \hat{\mathbf{c}}$ via expensive matrix-vector multiplications involving all of the non-zero entries of matrices A, B and C.

In this section, we explain how to construct a *holographic* polynomial IOP protocol for R1CS, in which the verifier's direct access to A, B and C is replaced by query access, as in [CHM+20; COS20]. In this construction, the prover can either run in linear-time and linear-space, or quasilinear-time and log-space, and the verifier runs in log-time and log-space.

Theorem 3 (informal)**.** *There exists an elastic* **holographic** *polynomial IOP for $\mathcal{R}_{\mathrm{R1CS}}$, whose prover admits two implementations:*

– *the* **time-efficient** *prover runs in $O(M)$ time and $O(M)$ space;*
– *the* **space-efficient** *prover runs in $O(M \log^2 M)$ time and $O(\log M)$ space,*

where N is the size of the R1CS input, and M is the number of non-zero entries in the R1CS instance. The verifier runs in $O(|\mathbf{x}| + \log M)$ time and space.

High-level overview. Our holographic protocol follows the same strategy as prior work [BCG20]. Roughly speaking, the core difference between the holographic protocol in this section and the non-holographic protocol in Section 2.5 is that the prover and verifier use an alternative strategy to check Equation 5. Instead of reducing Equation 5 to Equation 6, and then verifying Equation 6 via $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$ and $\hat{\mathbf{c}}$, the prover sends extra oracle messages to the verifier, corresponding to partial computations of Equation 5. Then, the prover and the verifier engage in various sub-protocols to check that the partial computations were performed correctly. As in prior works [BCG20], the key sub-protocols are a *look-up* protocol and an *entry-product* protocol (also known as grand product argument [Set20]).

Our main contribution is a space-efficient realisation of these sub-protocols, which leads to a space-efficient holographic R1CS protocol. The main challenge is to show that it is possible to generate the prover's extra messages in a space-efficient manner from a streaming R1CS instance (Definition 2). This places particular restrictions on the design of a space-efficient look-up protocol, which we explain how to deal with in Section 2.6.1. We explain how to construct a space-efficient entry-product protocol in Section 2.6.2.

Achieving holography. For a matrix $U \in \{A, B, C\}$, consider the vectors row, col, $\mathsf{val} \in \mathbb{F}^M$, such that, for each $i \in [M]$, $\mathsf{val}_i \in \mathbb{F}$ is the $(\mathsf{row}_i, \mathsf{col}_i)$-entry of U, ordered column-major. In the construction of a holographic PIOP, we will assume that the matrices A, B and C have the same support, which means that $\mathsf{row} := \mathsf{row}_A = \mathsf{row}_B = \mathsf{row}_C$ and $\mathsf{col} := \mathsf{col}_A = \mathsf{col}_B = \mathsf{col}_C$. This can easily be achieved by padding val_A, val_B and val_C with zeroes as required, and increases the length of row, col and val by at most a factor of 3.

The prover constructs the following vectors and sends them to the verifier as oracle messages:

$$\mathbf{r}^*_A := \mathbf{y}_A|_{\mathsf{row}} \ , \qquad \mathbf{r}^*_B := \mathbf{y}_B|_{\mathsf{row}} \ , \qquad \mathbf{r}^*_C := \mathbf{y}_C|_{\mathsf{row}} \ , \qquad \mathbf{z}^\star := \mathbf{z}|_{\mathsf{col}} \ . \tag{9}$$

In Equation 9, $\mathbf{r}^*_A$ is the vector whose i-th element is the (row_i)-th element of $\hat{\mathbf{a}}$, and similarly for $\mathbf{r}^*_B$, $\mathbf{r}^*_C$ and $\mathbf{z}^\star$. Using Equation 9, Equation 4 can be reformulated as:

$$\begin{aligned} \langle \mathbf{r}^*_A \circ \mathsf{val}_A,\ \mathbf{z}^\star \rangle &= u_A \ , \\ \langle \mathbf{r}^*_B \circ \mathsf{val}_B,\ \mathbf{z}^\star \rangle &= u_B \ , \\ \langle \mathbf{r}^*_C \circ \mathsf{val}_C,\ \mathbf{z}^\star \rangle &= u_C \ . \end{aligned} \tag{10}$$

Then, the verifier must check the three claims of Equation 10, and that $\mathbf{r}^*_A$, $\mathbf{r}^*_B$, $\mathbf{r}^*_C$ and $\mathbf{z}^\star$ were correctly computed. The prover and verifier run a twisted scalar-product protocol for the three claims. To check that $\mathbf{r}^*_A$, $\mathbf{r}^*_B$, $\mathbf{r}^*_C$ and $\mathbf{z}^\star$ were correctly computed, the prover and verifier run a look-up protocol, which we describe in more detail in Section 2.6.1.

Elastic realization. The twisted scalar-product protocol and look-up protocol are elastic protocols with both time and space-efficient prover realization, and a succinct verifier. Our holographic protocol for R1CS inherits a time-efficient prover and succinct verifier from these sub-protocols. However, to give a space-efficient prover realisation, we must show that the prover can produce streams of $\mathbf{r}^*_A$, $\mathbf{r}^*_B$, $\mathbf{r}^*_C$, using input R1CS streams and the verifier challenges. The R1CS streams $\mathcal{S}_{\rm col}(A)$, $\mathcal{S}_{\rm col}(B)$ and $\mathcal{S}_{\rm col}(C)$ of the matrices A, B and C produce elements of the form $(i, j, e) \in [N] \times [N] \times \mathbb{F}$. Streaming only the first element of the triple produces the stream $\mathcal{S}_{\rm cmrow}(A) = \mathcal{S}_{\rm cmrow}(B) = \mathcal{S}_{\rm cmrow}(C)$ of the vector row (we recall that we assumed the support of A, B, C to be the same, and that row is ordered column-major).

Similairly, the second element of the triple induces a stream $\mathcal{S}_{\rm cmcol}(A)$ of the vector col, which is also equal to $\mathcal{S}_{\rm cmcol}(B)$ and $\mathcal{S}_{\rm cmcol}(C)$, again since the support is the same. Additionally, $\mathcal{S}_{\rm cmcol}(A)$ is non-increasing: the column indices, in the dense representation of the matrix, are sorted in decreasing order when streamed column-major. As a result, the entries of $\mathbf{z}^\star$ can be produced one by one in $O(1)$ space from streams $\mathcal{S}(\mathbf{z})$ and $\mathcal{S}_{\rm col}(A)$: examine each entry of $\mathcal{S}_{\rm cmcol}(A)$, advance forwards $\mathbf{z}$ if the column changed, and output that same entry as long as the next element of $\mathcal{S}_{\rm cmcol}(A)$ stays unchanged.

The streams $\mathcal{S}_{\rm cmval}(A)$ (respectively, $\mathcal{S}_{\rm cmval}(B)$ and $\mathcal{S}_{\rm cmval}(C)$) are defined by projecting onto the third element of the streams $\mathcal{S}_{\rm col}(A)$ (respectively, $\mathcal{S}_{\rm col}(B)$ and $\mathcal{S}_{\rm col}(C)$), and produce the streams for the vectors val_A, val_B, and val_C in column-major order.

For $\mathbf{r}^*_A$, $\mathbf{r}^*_B$ and $\mathbf{r}^*_C$, recall that $\mathbf{y}_B = \otimes_j(1, \rho_j)$, $\mathbf{y}_C = \otimes_j(1, v^{2^j})$, and $\mathbf{y}_A = \mathbf{y}_B \circ \mathbf{y}_C = \otimes_j(1, \rho_j v^{2^j})$. Thus, any entry of $\mathbf{r}^*_B$ or $\mathbf{r}^*_C$ (and hence $\mathbf{r}^*_A$) can be computed in $O(\log N)$ operations from $v \in \mathbb{F}^\times$ and $\rho_0, \ldots, \rho_{n-1} \in \mathbb{F}^\times$.

2.6.1 Lookup protocol

Lookup protocols enable the prover to convince the verifier that all of the entries in a vector $\mathbf{g}^* \in \mathbb{F}^M$ appear as entries of another vector $\mathbf{g} \in \mathbb{F}^N$ according to the data stored in the *address vector* $\mathsf{addr} \in [N]^M$, i.e.:

$$\{(\mathbf{g}^*_i, \mathsf{addr}_i)\}_{i\in[M]} \subseteq \{(\mathbf{g}_j, j)\}_{j\in[N]} \ .$$

We denote this condition by "$(\mathbf{g}^*, \mathsf{addr}) \subseteq (\mathbf{g}, [N])$". In order to verify that $\mathbf{r}^*_U$ and $\mathbf{z}^\star$ were correctly computed, the verifier must check four lookup relations:

$$\begin{array}{cc} (\mathbf{r}^*_A, \mathsf{row}) \subseteq (\hat{\mathbf{a}}, [N]) \ , & (\mathbf{r}^*_B, \mathsf{row}) \subseteq (\hat{\mathbf{b}}, [N]) \ , \\ (\mathbf{r}^*_C, \mathsf{row}) \subseteq (\hat{\mathbf{c}}, [N]) \ , & (\mathbf{z}^\star, \mathsf{col}) \subseteq (\mathbf{z}, [N]) \ . \end{array} \tag{11}$$

Note that since $\mathbf{y}_A = \mathbf{y}_B \circ \mathbf{y}_C$, and $\mathbf{r}^*_A$, $\mathbf{r}^*_B$ and $\mathbf{r}^*_C$ come from looking up the entries of $\mathbf{y}_A$, $\mathbf{y}_B$ and $\mathbf{y}_C$ at the indices specified by row. Therefore, instead of checking that $(\mathbf{r}_A^*, \mathsf{row}) \subseteq (\mathbf{y}_A, [N])$, it suffices to check the Hadamard product relation $\mathbf{y}_A = \mathbf{y}_B \circ \mathbf{y}_C$. This can be done using an extension of the twisted scalar product protocol. This leaves four look-up relations to check.

Polynomial identities for look-up relations. To verify look-up relations, we use the polynomial identity derived by Gabizon and Williams [GW20], and similar strategies to Bootle et al. [BCG20] to construct a polynomial IOP to verify the identity.

We reduce the lookup conditions $\big(\mathbf{r}_U^*, \mathsf{row}\big) \subseteq \big(\mathbf{y}_U, [N]\big)$ and $\big(\mathbf{z}^\star, \mathsf{col}\big) \subseteq \big(\mathbf{z}, [N]\big)$ to simpler inclusion conditions such as $\mathbf{f}^* \subseteq \mathbf{f}$, where each entry in the vector $\mathbf{f}^*$ equals some entry in the vector $\mathbf{f}$. To do so, for each matrix $U = A, B, C$, *algebraically hash* the pairs $(\mathbf{r}_U^*, \mathsf{row})$, $(\mathbf{y}_U, [N])$, $(\mathbf{z}^\star, \mathsf{col})$ and $(\mathbf{z}, [N])$ into single vectors by considering a random linear combination of each pair, using a random challenge from the verifier. Let $\mathsf{sort}(\mathbf{g}, \mathbf{f})$ denote the function that sorts the entries of $\mathbf{g} \;||\; \mathbf{f}$ according to order of appearance in $\mathbf{f}$.

Lemma 3 ([GW20, Claim 3.1]). *Let $\mathbf{f}^* \in \mathbb{F}^M$ and $\mathbf{f} \in \mathbb{F}^N$. Then $\mathbf{f}^* \subseteq \mathbf{f}$ if and only if there exists $\mathbf{w} \in \mathbb{F}^{M+N}$ such that the equation below in $\mathbb{F}[X, Y]$ is satisfied:*

$$\prod_{j=0}^{M+N-1} \big(Y(1+Z) + w_{j+1} + w_j \cdot Z\big) = \\ (1+Z)^M \prod_{j=0}^{M-1} (Y + f_j) \prod_{j=0}^{N-1} \big(Y(1+Z) + f_{j+1} + f_j \cdot Z\big) \quad (12)$$

where indices are taken (respectively) modulo $M + N$, N. If $\mathbf{f}^* \subseteq \mathbf{f}$, then $\mathbf{w} := \mathsf{sort}(\mathbf{f}^*, \mathbf{f})$ satisfies Equation 12.

The strategy in the look-up protocol is for the prover to compute $\mathbf{w}$ and prove that Equation 12 is satisfied for every look-up relation that needs to be checked. In the protocol, the prover computes $\mathbf{w}$ and sends it to the verifier. Then, the verifier sends random challenges $\upsilon, \zeta \in \mathbb{F}^\times$ to the prover, who computes each of the three product expressions in Equation 12, evaluated at υ and ζ:

$$\begin{aligned} e_0 &= \prod_{i=0}^{M+N-1} \big(\upsilon(1+\zeta) + w_{i+1} + w_i \cdot \zeta\big), \\ e_1 &= \prod_{i=0}^{M-1} (\upsilon + f_i^*), \\ e_2 &= \prod_{i=0}^{N-1} \big(\upsilon(1+\zeta) + f_{i+1} + f_i \cdot z\big) \;\; . \end{aligned} \quad (13)$$

where (again) indices are taken (respectively) modulo $M + N$, N. The prover then sends the three product values e_0, e_1 and e_2 to the verifier. The verifier checks Equation 12 holds at υ and ζ by checking that $e_0 = (1+\zeta)^M e_1 e_2$, and uses three *entry-product* sub-protocols, which we describe in Section 2.6.2, to prove that e_0, e_1 and e_2 were correctly computed from $\mathbf{f}^*$, $\mathbf{f}$ and $\mathbf{w}$.

This approach requires polynomial query access to $\mathbf{f}^*_{\circlearrowright}$, the cyclic right-shift of $\mathbf{f}^*$, since the inputs to the entry product protocols depend on $\mathbf{f}^*_{\circlearrowright}$. The look-up protocol of Bootle et al. [BCG20] uses an additional *shift* sub-protocol to check this condition. By contrast, we remove this additional step by considering instead the lookup protocol over vectors with a leading zero coefficient. Now, queries on the right-shift $\mathbf{f}^*_{\circlearrowright}$ can be related to queries on $\mathbf{f}^*$ with a single evaluation query, since the leading coefficient is known in advance. We explain this optimisation further in the full version of this paper.

Elastic realization. As shown in prior work [BCG20], if the underlying entry product protocols have a linear-time prover realisation and succinct verifier, then the same is true for the look-up protocol. Therefore, we focus on explaining a space-efficient prover realisation of the look-up protocol. Assuming that the entry-product protocol has a suitable space-efficient realisation, it suffices to explain how to simulate streaming access to look-up protocol vectors $\mathbf{f}^*$, $\mathbf{f}$ and $\mathbf{w}$ using previously derived streams.

First we consider $(\mathbf{z}^\star, \mathsf{col})$ and $(\mathbf{z}, [N])$. Recall that each pair is algebraically hashed into vectors $\mathbf{f}^*$ and $\mathbf{f}$. It is simple produce the streams $\mathcal{S}(\mathbf{f}^*)$ and $\mathcal{S}(\mathbf{f})$ from the streams $\mathcal{S}(\mathbf{z}^\star)$, $\mathcal{S}_{\text{cmcol}}(A)$, $\mathcal{S}(\mathbf{z})$ and $\mathcal{S}([N])$, by applying the same algebraic hash function to pairs of entries on-the-fly. The same applies to input pairs $(\mathbf{r}^*_U, \mathsf{row})$ and $(\mathbf{y}_U, [N])$.

Now, we explain how to generate a stream of $\mathbf{w} = \mathsf{sort}(\mathbf{f}^*, \mathbf{f})$ using little memory space. This is more challenging because storing the entire vectors $\mathbf{f}^*$ and $\mathbf{f}$ and sorting them requires $O(M + N)$ memory. In the case of inputs $(\mathbf{z}^\star, \mathsf{col})$ and $(\mathbf{z}, [N])$, as col is a non-decreasing sequence, it turns out that $\mathcal{S}_{\text{cmcol}}(A)$ is already sorted into a suitable order, and it suffices to *merge* the streams of $\mathbf{f}^*$ and $\mathbf{f}$ together to produce a stream for $\mathbf{w}$. The same can't be said for row, which is not necessarily ordered. However, the vector row in non-decreasing form is already available from the inputs: it can be streamed from the dense representation of the matrix in row-major ordering $\mathcal{S}_{\text{row}}(A)$. To apply the same idea to input pairs $\mathbf{r}^*_U$ and row, we build $\mathcal{S}_{\text{rmrow}}(A)$, which is non-decreasing, and use it to produce the stream of the sorted vector for the lookup protocol. We describe our look-up protocol in more detail in the full version.

On alternative proof techniques for look-up relations. Prior work such as [Set20] checks look-up relations using an *offline memory-checking* [BEG+91; CDD+03] abstraction in which the prover shows that $\mathbf{g}^*$ was correctly constructed entry by entry from $\mathbf{g}$ using read and write operations. This leads to an alternative polynomial identity replacing Equation 12, which uses a list of *timestamps* recording when a particular element of $\mathbf{g}^*$ was read from $\mathbf{g}$. In this case though, it is unclear how to generate the timestamps required for by this method without storing linear memory. While in the argument of Gabizon and Williams [GW20] the polynomial relation is independent from the ordering of the matrix $\mathcal{S}(A)$ (row- or column-major), memory-checking arguments require random access to the vector row in order to access the last visited timestamps, which cannot be performed in log-space.

2.6.2 Entry product protocol

Let $\mathbf{f} = (f_0, \dots, f_{N-1}) \in \mathbb{F}^N$ such that $e = f_0 \cdots f_{N-1}$. We describe an entry-product protocol, building on Bootle et al. [BCG20, Sec. 6.4], that reduces an entry product statement $\prod_i f_i = e$ to a single scalar-product relation, using polynomial evaluation query access to $\mathbf{f}$.

Compared with the prior work, our work exploits the structure of univariate polynomials to simplify the scheme and remove the need for *cyclic-shift tests* [BCG20, Sec. 6.3]. We propose additional optimizations in Section 2.7 which improve the concrete efficiency of our protocol.

High-level overview. Let $\mathbf{f}$ be as above, with $f_{N-1} = 1$.[11] Let $\psi \in \mathbb{F}^\times$ and let $\mathbf{y}' = (1, \psi, \dots, \psi^{N-1})$. Let $\mathbf{g}$ be the vector of partial products of the entries of $\mathbf{f}$, that is:

$$\mathbf{g} \coloneqq \left(\textstyle\prod_{i \geq 0} f_i, \quad \prod_{i \geq 1} f_i, \quad \dots, \quad f_{N-2}f_{N-1}, \quad f_{N-1}\right) \tag{14}$$

Then, observe that:

$$\begin{aligned} \langle \mathbf{g} \circ \mathbf{y}', \mathbf{f}_{\circlearrowleft} \rangle &= \sum_{i=1}^{N-1} g_i f_{i-1} \psi^i + g_0 f_{N-1} \\ &= \sum_{i=1}^{N-1} g_{i-1} \psi^i + e + g_{N-1}\psi^N - g_{N-1}\psi^N \\ &= \psi \mathbf{g}(\psi) + e - \psi^N \end{aligned} \tag{15}$$

In the entry product protocol, the prover sends the oracle $\mathbf{g}$ to the verifier, and the verifier replies with the random challenge $\psi \in \mathbb{F}^\times$, and makes a polynomial evaluation query $\mathbf{g}(\psi) = v$. Then, both parties engage in a twisted scalar product protocol to verify Equation 15. Polynomial evaluation queries $\mathbf{f}_{\circlearrowleft}(x)$ for $x \in \mathbb{F}$ made as part of the twisted scalar-product protocol can be computed using evaluation queries $\mathbf{f}(x)$. To do this, note that $\mathbf{f}_{\circlearrowleft}(x) = x\mathbf{f}(x) - x^N + 1$ since $f_{N-1} = 1$; thus the verifier can compute $\mathbf{f}_{\circlearrowleft}(x)$ from $\mathbf{f}(x)$ in $O(\log N)$ operations. The partial products in Equation 14 are computed starting with f_{N-1} because

Elastic realization. As with other sub-protocols, the entry-product protocol inherits a linear-time prover realisation and succinct verifier from the underlying twisted scalar-product protocol.

To give a space-efficient realisation, it suffices to show that $\mathbf{g}$ can be generated element-by-element given access to the stream $\mathcal{S}(\mathbf{f})$: the partial products of

[11] This restriction is merely didactical. Given any $\mathbf{f} \in \mathbb{F}^N$, representing the coefficients of a degree $N-1$ polynomial, it is easy to simulate polynomial-evaluation query access to $(\mathbf{f}, 1)$ using the polynomial $\mathbf{f}(X) + X^{N+1}$. For any evaluation query in $x \in \mathbb{F}$, forward evaluation queries to $\mathbf{f}$ and add x^{N+1} before returning. This costs $O(\log N)$ $\mathbb{F}$-ops.

elements of $\mathbf{f}$ can be produced by streaming each successive element of $\mathcal{S}(\mathbf{f})$ and multiplying it into a running product. Note that the partial products in $\mathbf{g}$ are computed from the last entry to the first, starting with f_{N-1}. This is because streams of polynomials move from the highest-order coefficient to the lowest to be compatible with space-efficient commitment algorithms, as explained in Section 2.3.

2.7 Implementation and optimizations

We implemented the elastic argument from Sections 2.5 and 2.6 by leveraging and extending `arkworks` [ark], a Rust ecosystem for developing and programming with zk-SNARKs. Our implementation, called `ark-gemini`[12], is open-source and freely available under MIT license. The code structure follows the modular design of the protocol, which involves combining an elastic polynomial commitment scheme and an elastic (holographic) PIOP. We deem each of the single components of the protocol (the streaming infrastructure, the commitment scheme, and the sub-protocols for sumcheck, tensor check, entry product, lookup protocol, etc.) to be independent interest for future space-efficient projects. Below, we provide an overview of the streaming infrastructure and the algorithmic optimizations that were adopted in the implementation.

2.7.1 Streaming infrastructure

We extend the `arkworks` framework with support for streams in order to express our space-efficient protocols. A stream is simply a wrapper over `iter::Iterator`, the Rust interface for dealing iterators. Streams can be restarted and iterated over multiple times. We use Rust's borrow abstractions to produce streams that avoid copying elements whenever possible: a stream either returns a field element, or a reference to a field element. In other words, we have zero-copy interface where data structures do not require to be copied from memory, unless really needed. In practice, input streams could be instantiated with arrays (for instance, a memory-mapped files), or a concurrent stream of data downloaded from the web. Our design supports stream compositions and could be potentially extended to new front-ends.

A recent work by Baum, Malozemoff, Rosen, and Scholl [BMRS21] also studies streaming provers, and provides a space-efficient proving algorithm in Rust. To achieve a space-efficient prover, they rely on Rust's concurrency features (also known as Rust async), which is a more specific interface compatible with our framework based on iterators.

2.7.2 Practical optimizations

We introduce several algorithmic optimizations that improve the concrete performance of our scheme.

Elastic provers. One of the benefits of the elastic SNARK is that it allows switching from the space-efficient implementation to the time-efficient one. For example, in the scalar product, if the prover has enough memory, then it can

[12] See `https://github.com/arkworks-rs/gemini`

transcribe the folded sumcheck claim and proceed with the time-efficient implementation of the prover function. This allows for a more fine-grained control of the memory for the prover, and benefit from the speed-up of the time-efficient prover for the last few rounds of the protocol. Since the prover's messages are the same in both modes, this does not affect the end result. In our implementation, it is possible to enforce a memory budget that, once hit, allows the prover to stop and store the intermediate claim entirely in memory. Once the claim has been stored, it is possible to proceed with the time-efficient implementation.

Batch [KZG10] . Boneh et al. [BDFG20] proposed an optimization of [KZG10] to batch evaluation proofs for a set of evaluation points over different polynomials, exploiting the special structure of univariate polynomials.

We adapt these optimizations to our elastic polynomial commitment scheme, and implement them. In particular, although our tensor product protocol may require the verifier to query different polynomials at a set of different evaluation points, a single constant-size evaluation will have to be sent. This renders the concrete size of the proof significantly better than multi-linear approaches such as [ZGK+17; ZGK+18], which require a logarithmic-size opening proof.

Offline memory-checking. As discussed in Section 2.6.1, the offline memory-checking protocol is not compatible with the space-efficient prover, because the computation of timestamps may require random-access over non-zero entries. However, we also observe that given a particular ordering of the non-zero entries, it might be possible to apply the offline memory-checking partially in our polynomial IOP. In the particular implementation of our protocol, the offline memory checking can be used to prove the lookup for $\big(\mathbf{z}^{\star}_{U}, \mathsf{col}_{U}\big) \subseteq \big(\mathbf{z}, [N]\big)$.

We view the offline memory-checking as an optimization because it is concretely more efficient than the plookup protocol. That is because the sender in the plookup protocol must send additional commitments to the verifier; whereas, the commitments in the offline memory-checking can be precomputed by the indexer.

2.8 Evaluation

We run extensive benchmarks over Gemini (both preprocessing and non-preprocessing SNARKs), over an Amazon AWS EC2 `c5.9xlarge` instance, with 36 cores. We enable multi-threading using `rayon`, a Rust library for parallelism, and use it for efficiently computing multi-scalar multiplications and to run the batched sumcheck in the preprocessing SNARK protocol, where multiple sumcheck instances can be run in parallel. We select BLS12-381 as the pairing-friendly elliptic curve, but we note however that smaller elliptic curves are suitable, due to the remark in Remark 4. Our chosen field $\mathbb{F}$ is the scalar field of BLS12-381.

We perform tests for different instance sizes N, with $M = N$, for the range $N = 2^{18}$ up to 2^{35}. These instance sizes are much larger than what is commonly covered in the literature, and are meant to illustrate the behavior of a proving system over very large instances.

Proving space. We show that the Gemini prover can support instances with arbitrary sizes. In Fig. 2 it is possible to observe that the memory trace remains constant across large instance sizes, and that is stays consistently below 1GB of required memory, while the preprocessing SNARK protocol demands slightly more than 1GB to run. Two main constants influence the overall memory trace of the program:

– the memory budget allocated for multi-scalar multiplication (MSM). Despite the MSM operation can also be implemented in a streaming fashion, either with trivial scalar-multiplication, or by making small changes over Pippenger's algorithm, we noted that, in practice, best performance is achieved by performing Pippenger over buffers of fixed sizes, and then accumulating the partial result. We set the buffer to host 2^{20} field elements.
– the sumcheck round threshold, after which the elastic prover will transcribe the sumcheck instance and proceed with the time-efficient algorithm. We set the threshold to 22. That is, the last 22 rounds of the sumcheck will always be performed with the time-efficient prover.

The memory footprint stays constant because the constants chosen for the multi-scalar multiplication buffer and sumcheck round threshold are much larger than the asymptotic factors of the proving algorithm. The difference in memory is related solely to the batched sumcheck step in the preprocessing SNARK protocol, where multiple instances are being transcribed in memory at the same time once the round threshold hits.

Our benchmarks stop at 2^{35} for the non-preprocessing SNARK and the 2^{32} for the preprocessing one, but the upper limit in our benchmarks is arbitrary: as long as it is possible to generate the input streams for the time prover, then prover will be able to carry out in full the proving algorithm, and keeping the memory footprint very small. Prior works, such as Setty [Set20] and Chiesa et al. [CHM+20] provide public benchmarks for sizes up to 2^{20}. When running benchmarks ourselves to compare our work with previous literature such as Marlin[13], we were unable to proceed over size 2^{24} due to out of memory crashed. Even if we instruct the kernel to allow for memory over-commitment[14], the kernel will refuse to allocate new memory and eventually Rust will panic due to memory allocation failures. Our own prover, when run as a purely time-efficient prover, cannot successfully prove instances of size 2^{25} and 2^{27}.

Proving time. We present the proving time of elastic provers. The elastic prover will switch to the time-efficient mode if the intermediate state can be loaded within the memory budget. So when the instance size is small, the elastic prover will run purely in the time-efficient mode. As far as runtime is concerned, we make an important initial remark: the most expensive operations in our protocol are given by the cryptographic operations, namely the multi-scalar multiplications. For this reason, in Fig. 2, where we show the running times for for different values of N, with $M = N$, it is possible to observe a graph that evolves almost linearly. The squared logarithmic factor does not influence

[13] cf. `https://github.com/arkworks-rs/marlin`.
[14] This is `vm.overcommit=2`. See `https://www.kernel.org/doc/Documentation/vm/overcommit-accounting`.

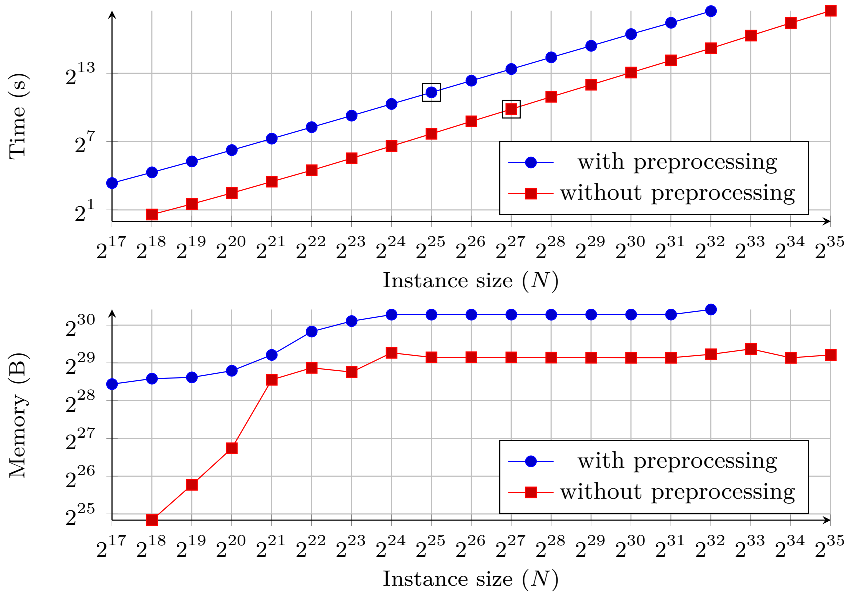

Fig. 2: Runtimes (above) and memory usage (below) for the elastic prover in the *preprocessing* protocol (blue) and the *non-preprocessing* protocol (red), for different R1CS sizes with $N = M$. The black squares indicate the size for which the time-efficient prover triggers an out-of-memory crash.

noticeably the overall runtime, as far as we were able to measure within the window of instance sizes of our benchmarks.

We also measure the economic cost of running the Gemini prover. Roughly speaking, the cost per gate of the preprocessing SNARK prover is around than 7.6×10^{-5} seconds per gate. Using the AWS estimator[15] (on-demand hourly cost 1.836 USD), we are able to conclude that, roughly speaking, the cost for the preprocessing SNARK is about 2.30×10^{-5} USD per gate. In particular, the estimated cost for an instance of 2^{31} gates is 89 USD. In contrast, the cost of DIZK [WZC+18] is much higher and around 500 USD for an instance of 2^{31}, because DIZK has to run the computation on 20 more powerful and expensive machines (`r3.8xlarge` EC2 instances with on-demand hourly cost 2.656 USD) for about 10 hours. In the case of non-preprocessing SNARK, the cost is a bit lower and around 40 USD for a circuit of 2^{35}.

Verification time and proof size. We measure the proof size and verification time for the preprocessing protocol. Note that the verifier can easily verify the proof for large instances since it does not need to read and load the instance into the memory. For instance size ranging from 2^{12} to 2^{35}, the proof size is about $13 - 27$ KB, and the verification time is about $16 - 30$ ms.

[15] source: https://calculator.aws

References

[ark] arkworks. *arkworks: an ecosystem for developing and programming with zkSNARKs.* URL: `https://github.com/arkworks-rs`.

[BBHR18] E. Ben-Sasson et al. "Fast Reed–Solomon Interactive Oracle Proofs of Proximity". In: ICALP '18.

[BC12] N. Bitansky et al. "Succinct Arguments from Multi-Prover Interactive Proofs and their Efficiency Benefits". In: CRYPTO '12.

[BCC+09] M. Belenkiy et al. "Randomizable Proofs and Delegatable Anonymous Credentials". In: CRYPTO '09.

[BCC+16] J. Bootle et al. "Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting". In: EUROCRYPT '16.

[BCCT13] N. Bitansky et al. "Recursive Composition and Bootstrapping for SNARKs and Proof-Carrying Data". In: STOC '13.

[BCG+14] E. Ben-Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: SP '14.

[BCG+17] J. Bootle et al. "Linear-Time Zero-Knowledge Proofs for Arithmetic Circuit Satisfiability". In: ASIACRYPT '17.

[BCG+18] J. Bootle et al. "Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution". In: ASIACRYPT '18.

[BCG20] J. Bootle et al. "Linear-Time Arguments with Sublinear Verification from Tensor Codes". In: TCC '20.

[BCR+19] E. Ben-Sasson et al. "Aurora: Transparent Succinct Arguments for R1CS". In: EUROCRYPT '19.

[BCS16] E. Ben-Sasson et al. "Interactive Oracle Proofs". In: TCC '16-B.

[BCTV14] E. Ben-Sasson et al. "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture". In: USENIX Security '14.

[BDFG20] D. Boneh et al. "Efficient polynomial commitment schemes for multiple points and polynomials". Cryptology ePrint Archive, Report 2020/081.

[BEG+91] M. Blum et al. "Checking the correctness of memories". In: FOCS '91.

[BFS20] B. Bünz et al. "Transparent SNARKs from DARK Compilers". In: EUROCRYPT '20.

[BGM17] S. Bowe et al. "Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model". Cryptology ePrint Archive, Report 2017/1050.

[BHR+20] A. R. Block et al. "Public-Coin Zero-Knowledge Arguments with (almost) Minimal Time and Space Overheads". In: TCC '20.

[BHR+21] A. R. Block et al. "Time- and Space-Efficient Arguments from Groups of Unknown Order". In: CRYPTO '21.

[BMM+21] B. Bünz et al. "Proofs for Inner Pairing Products and Applications". In: ASIACRYPT '21.

[BMRS21] C. Baum et al. "Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions". In: CRYPTO '21.

[CDD+03] D. Clarke et al. "Incremental multiset hash functions and their application to memory integrity checking". In: Springer.

[CHM+20] A. Chiesa et al. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS". In: EUROCRYPT '20.

[CMT12] G. Cormode et al. "Practical Verified Computation with Streaming Interactive Proofs". In: ITCS '12.

[COS20] A. Chiesa et al. “Fractal: Post-Quantum and Transparent Recursive Proofs from Holography”. In: EUROCRYPT ’20.

[Dra20] J. Drake. “PLONK without FFTs”. URL: `https://www.youtube.com/watch?v=ffXgxvlCBvo`.

[DRZ20] V. Daza et al. “Updateable Inner Product Argument with Logarithmic Verifier and Applications”. In: ed. by A. Kiayias et al. Cham: Springer International Publishing. ISBN: 978-3-030-45374-9.

[FS86] A. Fiat et al. “How to prove yourself: practical solutions to identification and signature problems”. In: CRYPTO ’86.

[Gab20] A. Gabizon. “Lineval Protocol”. Available at `https://hackmd.io/aWXth2dASPaGVrXiGg1Cmg?view`.

[GGM14] C. Garman et al. “Decentralized Anonymous Credentials”. In:

[GGPR13] R. Gennaro et al. “Quadratic Span Programs and Succinct NIZKs without PCPs”. In: EUROCRYPT ’13.

[GLS+21] A. Golovnev et al. “Brakedown: Linear-time and post-quantum SNARKs for R1CS”. Cryptology ePrint Archive, Report 2021/1043.

[GW20] A. Gabizon et al. “plookup: A simplified polynomial protocol for lookup tables”. Cryptology ePrint Archive, Report 2020/315.

[HR18] J. Holmgren et al. “Delegating Computations with (Almost) Minimal Time and Space Overhead”. In: FOCS ’18.

[JW17] K. Javeed et al. “Low latency flexible FPGA implementation of point multiplication on elliptic curves over GF(p)”. In: *International Journal of Circuit Theory and Applications* (2017).

[KZG10] A. Kate et al. “Constant-Size Commitments to Polynomials and Their Applications”. In: ASIACRYPT ’10.

[LFKN92] C. Lund et al. “Algebraic Methods for Interactive Proof Systems”. In: *Journal of the ACM* (1992).

[PGHR13] B. Parno et al. “Pinocchio: Nearly Practical Verifiable Computation”. In: S&P ’13.

[Pip80] N. Pippenger. “On the Evaluation of Powers and Monomials”. In: (1980).

[PLS19] R. d. Pino et al. “Short Discrete Log Proofs for FHE and Ring-LWE Ciphertexts”. In: PKC ’19.

[RRR16] O. Reingold et al. “Constant-Round Interactive Proofs for Delegating Computation”. In: STOC ’16.

[Set20] S. T. V. Setty. “Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup”. In: CRYPTO ’20.

[Tha13] J. Thaler. “Time-Optimal Interactive Proofs for Circuit Evaluation”. In: CRYPTO ’13.

[WZC+18] H. Wu et al. “DIZK: A Distributed Zero Knowledge Proof System”. In: USENIX Security ’18.

[XZZ+19] T. Xie et al. “Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation”. In: CRYPTO ’19.

[Zcash] “Zcash”. `https://z.cash/`.

[ZGK+17] Y. Zhang et al. “vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases”. In: S&P ’17.

[ZGK+18] Y. Zhang et al. “vRAM: Faster Verifiable RAM with Program-Independent Preprocessing”. In: S&P ’18.

[ZWZ+21] Y. Zhang et al. “PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture”. In: ISCA ’21.