USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Hobbit: Space-Efficient zkSNARK with Optimal Prover Time

Christodoulos Pappas and Dimitrios Papadopoulos,
The Hong Kong University of Science and Technology

https://www.usenix.org/conference/usenixsecurity25/presentation/pappas

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

Hobbit: Space-Efficient zkSNARK with Optimal Prover Time

Christodoulos Pappas
HKUST

Dimitrios Papadopoulos
HKUST

Abstract

Zero-knowledge succinct non-interactive arguments (zkSNARKs) are notorious for their large prover space requirements, which almost prohibits their use for really large instances. Space-efficient zkSNARKs aim to address this by limiting the prover space usage, without critical sacrifices to its runtime. In this work, we introduce Hobbit, the *only existing space-efficient zkSNARK that achieves optimal prover time* $O(|\mathcal{C}|)$ for an arithmetic circuit $\mathcal{C}$. At the same time, Hobbit is the first transparent and plausibly post-quantum secure construction of its kind. Moreover, our experimental evaluation shows that Hobbit outperforms all prior general-purpose space-efficient zkSNARKs in the literature across four different applications (*arbitrary arithmetic circuits*, *inference of pruned Multi-Layer Perceptron*, *batch AES128 evaluation*, and *select-and-aggregate SQL query*) by $\times$**8**-$\times$**56** in terms or prover time while requiring up to $\times$**23** less total space.

At a technical level, we introduce two new building blocks that may be of independent interest: (i) the first sumcheck protocol for products of polynomials with optimal prover time in the streaming setting, and (ii) a novel multi-linear plausibly post-quantum polynomial commitment that outperforms all prior works in prover time (and can be tuned to work in a space-efficient manner). We build Hobbit by combining the above with a modified version of HyperPlonk, providing an explicit routine to stream access to the circuit evaluation.

1 Introduction

A *Zero-Knowledge Succinct Non-Interactive Argument of Knowledge* (zkSNARK) [1, 2] is a cryptographic protocol that enables a *prover* to convince a *verifier* about the validity of an NP statement, by producing a small and easy-to-verify proof that reveals nothing else other than the correctness of the statement. Nowadays, zkSNARKs have become integral components of securing many applications such as anonymizing cryptocurrencies [3], ensuring integrity of ML models [4, 5, 6], and verifying network traffic policies [7]. Although the main focus of zkSNARKs was initially to reduce the verification time and proof size, the cost of generating a proof has now become the main challenge. Over the past years, a substantial amount of progress has been made to reduce that cost resulting in constructions with concretely efficient and optimal prover times [8, 9, 10, 11, 12, 13, 14]. Despite these advancements, zkSNARKs still hardly scale for large instances due to *excessive prover space utilization*. In other words, the space used to prove a computation is multiple times larger than the one needed to natively evaluate it. For instance, to prove the correct hash computation for 2^{14} elements using the SHA256 function takes 8GB on Groth16 [8] and 32GB on Plonk [9]. At the same time, the space needed for the actual evaluation is no more than 40Kb.

Space-efficient zkSNARKs. Motivated by the above, an important research goal is constructing zkSNARKs that maintain low prover space. One way to achieve this is by developing *space-efficient* zkSNARKs which allow the prover to use space proportional to what is needed for the actual computation without sacrificing its runtime. At a very high level, all such zkSNARKs base their construction on the "*polynomial interactive oracle proof* (PIOP) + *polynomial commitment scheme* (PCS)" paradigm [15, 16] but adapt their prover to work in the *streaming setting*. In this setting, which was first formalized by Block et. al. [17] and adopted by all subsequent space-efficient arguments [5, 18, 19, 20, 21] the prover only has streaming access to the proving data, such as intermediate steps of the computation and public parameters (e.g., by generating them on the fly) and computes the proof using only a small *working buffer space*. Ideally, the total time and space of the prover (measured as the space needed to instantiate streaming access to the proving data plus its buffer space) should be proportional to the time and space needed for the computation, up to constant factors. Unfortunately, prior related work does not achieve this goal.

In particular, the first such zkSNARK was introduced by Block et al. [18] and incurred only a polylogarithmic blow-up in space and time compared to performing the computation. However, it suffers from very large concrete overheads for the prover's runtime, hence serving mostly as a theoretical

solution. Furthermore, it focuses on proving computations in the RAM model, which inherently introduces some additional bottlenecks to the prover time. Moving to the more efficient arithmetic circuit model, a very recent work, Sparrow [5], introduced a novel space-efficient PIOP based on the GKR protocol with $O(|\mathcal{C}|\log\log|\mathcal{C}|)$ prover time and space complexity of $O(S_{eval}+\sqrt{|\mathcal{C}|})$, where S_{eval} is the space needed to evaluate the circuit. Unfortunately, while Sparrow achieves an almost optimal prover time, it is restricted to proving the correctness of data-parallel arithmetic circuits.

To the best of our knowledge, the only space-efficient zkSNARKs that focus on general arithmetic circuits are Gemini [19] and Epistle [20]. In more detail, Gemini introduced a space-efficient PIOP for the R1CS system [11] along with a space-efficient variant of the univariate KZG PCS [22]. Likewise, Epistle constructed space-efficient variants of the HyperPlonk PIOP [13] and multi-variate KZG PCS [23]. Both schemes achieve a minimal working buffer space of $O(\log|\mathcal{C}|)$. In addition, they are elastic, which enables the prover to increase the working buffer space to $O(B)$ (where $B \geq \log|\mathcal{C}|$) with the benefit of concretely improving the prover time. However, both schemes consider a more "relaxed" streaming setting, leading to issues when it comes to measuring the *total space complexity*. In particular and contrary to all prior works [5, 17, 18, 21], they do not provide explicit constructions on how to instantiate streaming access to the proving data. Even worse, as we argue in Section 5, it is unclear how to achieve this without using $O(|\mathcal{C}|)$ space specifically for the constructions of [19, 20]. In addition, both schemes inherently suffer from $\Theta(|\mathcal{C}|)$ space complexity, due to the KZG PCS that requires public parameters of size $\Theta(|\mathcal{C}|)$. This is in contrast to the total space needed to evaluate a circuit, that may be asymptotically smaller for some circuits (e.g., log-space uniform [24] or data-parallel circuits [5]).

It is also worth mentioning that another approach of constructing zkSNARKs that maintain low prover space utilization is by "breaking-down" the computation in multiple smaller steps and recursively proving each one of them using *recursive proof composition* [25, 26]. However, compared to space-efficient zkSNARKs, this approach has two main disadvantages. First, they introduce additional overheads to the prover since they require recursively proving "SNARK-unfriendly" operations, such as hash functions, as part of proving the verifier logic. Second, efficient recursive proof composition approaches [6, 26, 27, 28, 29, 30, 31, 32] require the prover to make non-black use of random oracles, and hence, security cannot be formally argued, not even in the random oracle model (ROM). A recent line of works that try to overcome this issue [33, 34, 35], proposing analyses on novel (and not well-studied, for the time being) idealized models, such as AROM [33], but these results remain rather impractical.

Based on the above, and as Table 3 shows, we can make the following observations: *(1)* Unlike "standard" zkSNARKs, *there is no existing space-efficient zkSNARK with optimal prover time*. In fact, this is even true for space-efficient arguments that are non-succinct [21, 36]. *(2)* Existing space-efficient zkSNARKs in the arithmetic circuit model either require $\Theta(|\mathcal{C}|)$ space (independently of S_{eval}) or are restricted to specific circuit families. *(3)* No existing space-efficient zkSNARK is post-quantum secure, as all these schemes use PCS whose security is based on hardness problems over elliptic curve groups that are easy to solve by quantum adversaries.

Our work. Here, we introduce HOBBIT, the *first* space-efficient zkSNARK that overcomes all the above limitations. Our overall contributions can be summarized as follows:

1. We construct the first optimal-time space-efficient sumcheck for products of polynomials. For instances of size N, our scheme achieves $O(N)$ prover, working buffer space of $O(B)$, and $O(N/B+\log N)$ proof size and verification time for $B \in [\sqrt{N}, N]$ (Section 3).
2. We introduce a novel *plausibly* post-quantum secure, transparent, and linear time PCS with poly-logarithmic proof size, which outperforms all existing PC schemes with the same properties. We next adapt it in the streaming setting where we achieve $O(N)$ prover time, $O(B)$ working buffer space and $O(N/B+\log^2 N)$ proof size and verification time for $B \in [\sqrt{N}, N/\log N]$ (Section 4).
3. We combine the above to build HOBBIT, the first space-efficient zkSNARK with optimal prover time. It is also the first scheme that has transparent setup and is plausibly post-quantum secure. Compared to other space-efficient zkSNARKs, it supports lookup arguments, significantly improving its performance (Section 5).
4. We implemented and experimentally evaluated our constructions. Using only a single thread, HOBBIT can prove a circuit of size 2^{28}, in 1.1hrs using a total space of 4.2GB, which is only $\times 1.3$ larger than the space needed to evaluate the circuit. It also outperforms all existing space-efficient zkSNARKs [5, 19, 20] in both prover time and space. Our implementation is available at [37].

Furthermore, HOBBIT is *flexible*, i.e., it allows a trade-off between buffer space and proof size. Namely, one can increase B to reduce the proof size (or vice-versa, when working on restricted space settings). This is similar to elasticity, but trades off proof size rather than the prover time. In the full version, we also provide detailed proofs of security for all our constructions, following similar strategies as recent works [38, 39]. That is, we first prove their security in the interactive setting and then show that they satisfy *round-by-round* soundness [40, 41], a strengthening of the regular soundness that ensures the soundness of their non-interactive versions when applying the Fiat-Shamir transformation [40, 42, 43].

Technical Highlights. Next, we highlight some of the techniques that we developed towards our results.

Optimal-time & space-efficient sumcheck. The main building block of our PIOP is a sumcheck protocol which, given streaming access to the coefficients of the multi-linear polynomials $f, g : \mathbb{F}^{\log N} \to \mathbb{F}$, generates a proof for the instance: $K =$

Table 1: Asymptotic comparison of works on space-efficient arguments. For RAM programs, T is the number of execution steps, and S is the memory size. For an arithmetic circuit $\mathcal{C}$, S_{eval} denotes the space needed to evaluate it. x is the statement being proven by the argument. Finally, B is a threshold instance bounded between $\sqrt{|\mathcal{C}|}$ and $|\mathcal{C}|/\log|\mathcal{C}|$

Scheme	Model	Trans-parent	Plaus. Post Quantum	Prove	Verify	$\lvert\pi\rvert$	Total Space
Block et al. [17]	RAM	✔	✗	$T\cdot \text{polylog}\, T$	$T\log T$	$\log T$	$S\cdot \text{polylog}\, T$
Block et al. [18]	RAM	✔	✗	$T\cdot \text{polylog}\, T$	$\lvert x\rvert\cdot \text{polylog}\, T$	$\log T$	$S\cdot \text{polylog}\, T$
Ligetron [21]	WASM	✔	✔	$T\log T$	T	$\sqrt{T}$	$\sqrt{T}+S$
Gemini [19]	Arithm. Circ.	✗	✗	$\lvert\mathcal{C}\rvert\log^2\lvert\mathcal{C}\rvert$	$\lvert x\rvert+\log\lvert\mathcal{C}\rvert$	$\log\lvert\mathcal{C}\rvert$	$\lvert\mathcal{C}\rvert$
Epistle [20]	Arithm. Circ.	✗	✗	$\lvert\mathcal{C}\rvert\log\lvert\mathcal{C}\rvert$	$\lvert x\rvert+\log\lvert\mathcal{C}\rvert$	$\log\lvert\mathcal{C}\rvert$	$\lvert\mathcal{C}\rvert$
Sparrow [5]	LDP Arithm. Circ.	✔	✗	$\lvert\mathcal{C}\rvert\log\log\lvert\mathcal{C}\rvert$	$\lvert x\rvert+\log\lvert\mathcal{C}\rvert$	$\log\lvert\mathcal{C}\rvert$	$\sqrt{\lvert\mathcal{C}\rvert}+S_{eval}$
Our zkSNARK	Arithm. Circ.	✔	✔	$\lvert\mathcal{C}\rvert$	$\lvert x\rvert+\frac{\lvert\mathcal{C}\rvert}{B}+\log^2 B$	$\frac{\lvert\mathcal{C}\rvert}{B}+\log^2 B$	$B+S_{eval}$

$\sum_{\mathbf{x}\in\{0,1\}^{\log N}} f(\mathbf{x})g(\mathbf{x})$. Note that space-efficient sumcheck protocols already exist [5, 44] but either achieve $O(N\log N)$ prover time for $O(\log N)$ buffer space [44] or $O(N\log\log N)$ for $O(\sqrt{N})$ buffer space [5] or achieve linear prover time but are restricted to a sumcheck instance including a single polynomial [45], which is insufficient for constructing PIOPs. Instead, our protocol has optimal prover time. The main idea of achieving this is to partition the original sumcheck instance into N/B instances of size B and "aggregate" them in a streaming fashion (using a folding scheme for sumcheck instances) until reaching a single one where the prover has enough space to prove it using the "standard" sumcheck.

Although this idea looks simple, there are two subtle security concerns when trying to turn it into a zkSNARK: (1) Naively folding the sumchecks as done in [46]—by committing to each instance independently and then folding these commitments—would make the extractor for the original polynomials f and g grow exponentially with N, due to recursive extraction. We address this by directly committing to the original polynomials at the beginning (using our space-efficient PCS which we discuss next) and performing an additional streaming pass over f,g to reduce the evaluation claims of the folded polynomials to claims over f,g the validity of which is proven by directly opening the PCS. (2) When making our protocol non-interactive in the ROM via Fiat-Shamir, generic analysis [40, 41, 42] results in a "loose" bound and non-negligible error probability due to the large number of rounds. To mitigate this, we *explicitly* prove in our full version [37] that our protocol is round-by-round sound [40, 41], achieving a tighter bound that allows us to establish the soundness of its non-interactive variant in the ROM.

Optimal-time & space-efficient PCS. First, we propose a (space-inefficient) linear-time, plausibly post-quantum and transparent PCS with poly-logarithmic proof size and verification time. Although such schemes already exist, they either have impractically large proofs [12], slow provers [47], or are restricted to binary fields [38]. Even worse, it is unclear how to efficiently make some of them space-efficient [12].

In this work, we build upon the "Brakedown PCS [48] + Proof Composition" approach first appeared in [13] and subsequently improved by BrakingBase [47]. Unfortunately both schemes have prohibitively large prover times. This stems from the fact that when composing the proof, the prover has to run the encoding algorithm of a linear code for a long message. This leads to significant overheads because the only practical linear-time code to date is the Spielman code [49], which has a "SNARK-unfriendly" encoding algorithm.

Our main idea to solve this issue is that, instead of directly using a Spielman code as the underlying code, we can use a *tensor code* composed by a Spielman code and a "SNARK-friendly" code (e.g., an RS code [50]) configured in such way so that it has linear encoding time [51]. That allows us to reduce the cost of proof composition by exploiting the structure of the tensor code. Namely, the prover only has to "partially" compute the codeword using the SNARK-friendly linear code and then, by leveraging its linear property, invoke the Spielman encoding algorithm only once for a significantly smaller message size. Using this, we build our PCS and also show how to make it space-efficient.

Constructing HOBBIT. Armed with our space-efficient sumcheck and PCS, we can now build HOBBIT. To achieve this, we need to resolve two main challenges: (1) Construct a routine that computes all proving data on the fly using space $O(S_{eval})$ and (2) Develop a zkSNARK based on the data provided by this routine. For (1), we observe that an evaluation algorithm of $\mathcal{C}$ can be seen as a streaming algorithm where, in each computing step, either it deletes an existing gate or computes a new one. Based on this, we construct a routine that uses the evaluation algorithm to produce on the fly its *execution trace*, i.e., a record of the sequence of operations it performs. Given this routine, we build HOBBIT by adapting the PIOP of HyperPlonk [13] to work in our setting. The main challenge that arises is to prove consistency among wires in the circuit. In HyperPlonk, this is done using a permutation polynomial [9]. As we further elaborate in Section 5, however, it is unclear how to compute the coefficients of this polynomial on the fly using the data provided by our routine. Instead, we transform the problem of checking wiring consistency into proving the correctness of accesses from a memory that stores all gate values. Based on this, we adapt the memory checking technique of [11] to work in the streaming setting.

Implementation & Experimental Evaluation. We imple-

mented HOBBIT in C++ and evaluated its performance in four applications focused on proving the correct computation of: (a) arbitrary arithmetic circuits, (b) the inference of a pruned Multi-Layer Perceptron (MLP), (c) multiple AES128 block ciphers, and (d) SQL queries. To prove a circuit of size 2^{28}, HOBBIT takes 1.1hours (in a single thread) and utilizes a total space of 4.2GB. Note that the space needed to natively evaluate that circuit is 3.1GB. In other words, proving the correct computation of that circuit takes only $\times 1.3$ more space than what is required to evaluate it in the first place!

Compared to other space-efficient zkSNARKs for arithmetic circuits, HOBBIT is $\times 8.4$-$\times 32$ and $\times 54.32$-$\times 56.8$ faster than Gemini and Epistle and uses $\times 1.01$-$\times 23$ and $\times 1.1$-$\times 23.4$ less space. Even when we compare HOBBIT with the highly efficient Sparrow [5] on the applications corresponding to data-parallel circuits, we have $\times 1.5$-$\times 5$ faster prover and comparable space utilization. That is because, compared to Sparrow, our construction supports lookup arguments which can significantly reduce the size of the arithmetic circuit. The only drawback of HOBBIT is its larger proof size compared to other works, e.g., 6MB when proving 2^{15} AES128 block ciphers. That is expected as all existing schemes use elliptic curve-based PC schemes which offer significantly smaller proof sizes, with the cost of not being plausibly post-quantum.

Finally, we also compare the performance of our PCS, with existing plausibly post-quantum, transparent, and linear-time PC schemes. Overall, our PCS is $\times 3.1$-$\times 4.5$ and $\times 3.16$-$\times 3.6$ faster than BrakingBase and Orion respectively. Furthermore, our PCS offers smaller proof sizes and verification time. In addition, when compared to Brakedown [48] which has sub-linear proof size but an extremely fast prover, our PCS achieves $\times 1.03$-$\times 8.26$ smaller proofs and $\times 8.7$-$\times 29.7$ faster verifier with the cost of only $\times 2.6$ slower prover!

Other Related Works. Here we review other related works.
Other Space-Efficient Arguments. Block et al. [17] introduced a space-efficient argument of knowledge based on the PIOP of [44] which, although transparent, its PCS relies on Bulletproofs [52] and hence it has quasi-linear verifier. Based on [53, 54], Wang et al. [21] proposed Ligetron a space-efficient argument in the WASM model. While their construction is plausibly post-quantum and transparent, it has linear verification time and thus it is not succinct.
Disk-Based zkSNARKs. Recently, Baweja et al. [55] introduced Scribe, a *disk-based* zkSNARK. The main idea of Scribe is to reduce RAM utilization by storing and accessing all proving data or intermediate information produced by the prover throughout proof generation in a disk. Based on this, authors of Scribe show how to efficiently adapt Hyperplonk [13] in this setting. Note that Scribe is not space efficient as it inherently requires $O(|\mathcal{C}|)$ space—e.g., to store all proving data. Simply put, Scribe does not directly resolve the issue of excessive space utilization, but only mitigates it by moving the bulk of storage requirements to a "cheaper" (but potentially slower) storage medium.

2 Preliminaries

We denote $[n] = \{1,...,n\}$ and $\mathbb{F}$ a field of prime order p. We use $\mathbf{x} = (x_1,...,x_n) \in \mathbb{F}^n$ to represent a vector and $\mathbf{x}[i]$ its i-th element. Furthermore, for a matrix $\mathbf{A} \in \mathbb{F}^{n\times m}$, we denote with $\mathbf{A}[i:]$ and $\mathbf{A}[:i]$ is i-th row and column respectively. Finally, for an index i, j we use $\mathbf{i},\mathbf{j}$ to denote its bit-decomposition.

Multi-Linear Extensions. A *multi-linear extension* of a vector $\mathbf{x} : \{0,1\}^{\log n} \to \mathbb{F}$ is a multi-linear polynomial $f_\mathbf{x} : \mathbb{F}^{\log n} \to \mathbb{F}$, such that $f_\mathbf{x}(\mathbf{z}) = \sum_{\mathbf{i}\in\{0,1\}^{\log n}} \beta(\mathbf{z},\mathbf{i})\mathbf{x}[i]$, where $\beta(\mathbf{z},\mathbf{i})$ is the identity polynomial defined as $\prod_{k\in[\log n]} (\mathbf{i}[k]\mathbf{z}[k] + (1-\mathbf{z}[k])(1-\mathbf{i}[k]))$.

For all cryptographic components we describe next, we give formal definitions in the full version [37].

2.1 Polynomial Commitment Schemes

A *polynomial commitment scheme* (PCS) [22, 23] is a tuple of algorithms (*Gen*,*Commit*,*Open*,*Eval*,*Verify*) that enable a prover to commit to an n-variate polynomial of individual degree d (e.g., each variable has degree at most d), and later generate a proof showing that it correctly evaluated that polynomial at any random point. A PCS is *knowledge sound* if for any probabilistic polynomial time (PPT) adversary $\mathcal{A}_{PC}$ who generates an accepting proof that $f(\mathbf{x}) = y$, there exists an extractor $\mathcal{E}_{PC}$ that extracts the committed polynomial f, such that the probability that $f(\mathbf{x}) \neq y$ is negligible. Finally, a PC scheme is *zero-knowledge* if a verifier learns nothing more other than the correctness of the evaluation. In this work, we will construct a PCS for multi-linear polynomials (i.e., when $d = 1$) that is transparent, plausibly post-quantum and has optimal prover time. One common way to construct such PCS is to utilize linear codes [12, 47, 48].

Linear Codes. A linear error correction code is a linear subspace $C \in \mathbb{F}^n$ of dimension k. Every linear code has an injective mapping $Enc : \mathbb{F}^k \to C$ (called the encoding algorithm) that is given a message of size k and outputs a codeword in C. Furthermore, we denote with $\rho = k/n$ the rate of the code and δ the minumum relative distance defined as $min_{\mathbf{c}_1,\mathbf{c}_2}\{\Delta(\mathbf{c}_1,\mathbf{c}_2)/n\}$, for all $\mathbf{c}_1,\mathbf{c}_2 \in C$ such that $\mathbf{c}_1 \neq \mathbf{c}_2$ and $\Delta(\mathbf{c}_1,\mathbf{c}_2)$ their Hamming distance.

Brakedown PCS [48]. A PCS that leverages a linear code with linear encoding time [49] to achieve our desired properties is Brakedown. At a high level, it works as follows:
Commitment Algorithm. To commit $f : \mathbb{F}^{\log N} \to \mathbb{F}$, the prover first arranges its coefficients into a matrix $\mathbf{A}_f \in \mathbb{F}^{\sqrt{N}\times\sqrt{N}}$. Then, computes the *encoded matrix* $\mathbf{A}'_f \in \mathbb{F}^{\sqrt{N}\times(\sqrt{N}/\rho)}$, using the underlying encoding scheme for each row of $\mathbf{A}_f$. Finally, it commits to the columns of $\mathbf{A}'_f$ using a Merkle Tree.
Evaluation Algorithm. The evaluation algorithm consists of two phases: (a) proximity testing where the verifier ensures that the rows of the encoded matrix are close to valid codewords, and (b) consistency checking in which it validates

that $y = f(\mathbf{r})$ where $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2) \in \mathbb{F}^{(\log N)/2} \times \mathbb{F}^{(\log N)/2}$. For the proximity testing, the verifier samples a random vector $\mathbf{a} \in \mathbb{F}^{\sqrt{N}}$, sends it to the prover which replies with an aggregated vector $\mathbf{F}_1 = \sum_{i \in [\sqrt{N}]} \mathbf{a}[i]\mathbf{A}_f[i:]$. Then, the verifier computes $\mathbf{E}_1$, the codeword of $\mathbf{F}_1$, samples a column index set $I = \{i_1, \ldots, i_l\} \in [k\sqrt{N}]^l$ and sends it to the prover. Finally, it receives $\mathbf{R}^{\sqrt{N} \times l}$ consisting of the columns of $\mathbf{A}'_f$ at I along with their opening proofs. Upon validating their membership, it checks whether $\mathbf{E}_1[i_j] = \sum_{k \in [\sqrt{N}]} \mathbf{a}[k]\mathbf{R}_j[k], \forall i_j \in I$.

The consistency checking phase is almost identical to the proximity one but with the difference that the prover sends an aggregated vector $\mathbf{F}_2 = \sum_{i \in [\sqrt{N}]} \beta(\mathbf{i}, \mathbf{r}_1)\mathbf{A}_f[i:]$ and in the final verification step, the verifier also checks if $y = \sum_{i \in [\sqrt{N}]} \beta(\mathbf{i}, \mathbf{r}_2)\mathbf{F}_2[i]$. Finally, we note that both phases can run in parallel, using the same set I. The resulting PCS is transparent, plausibly post-quantum, and has optimal prover time. Unfortunately, its proof size and verification time are $O(\sqrt{N})$. Looking ahead, in Section 4 we show how to modify it to achieve poly-logarithmic proof size and verification time.

2.2 Interactive Proofs and PIOPs

An *Interactive Proof* (IP) is a protocol between a *prover* and *verifier* where both parties share an instance $\mathbf{x}$ and language $\mathcal{L}$ and the prover tries to convince the verifier that $\mathbf{x} \in \mathcal{L}$. We say that an interactive proof is *sound* if the verifier rejects with high probability when $\mathbf{x} \notin \mathcal{L}$ and *complete* if the verifier accepts whenever $\mathbf{x} \in \mathcal{L}$. A *Polynomial Interactive Oracle Proof* (PIOP) is an interactive proof with the additional property that the prover can reply with polynomial oracles. When receiving such an oracle, the verifier does not have to read its underlying polynomial entirely but only query their oracles at random points. Throughout the paper, we will make use of the following IP and PIOP protocols:

Sumcheck Protocol. Initially introduced by Lund et al. [56], the sumcheck protocol is an interactive proof in which both parties share a n-variate polynomial $f : \mathbb{F}^n \rightarrow \mathbb{F}$ of individual degree d, and the prover wishes to convince the verifier that $K = \sum_{\mathbf{x} \in \{0,1\}^n} f(\mathbf{x})$. The sumcheck protocol is perfectly complete, has a soundness error of at most $dn/|\mathbb{F}|$, $O(2^n)$ prover time and $O(n)$ communication complexity.

HyperPlonk Protocol. Proposed by Chen et al. [34], HyperPlonk is a PIOP for proving the correct execution of an arithmetic circuit $\mathcal{C}$. To encode a fan-in-2 circuit with addition and multiplication gates, HyperPlonk relies on the Plonkish arithmetization [9] that represents an arithmetic circuit via (1) selector functions and (2) a permutation function. At a high level, the selector functions which we denote with $add, mul : [s+N] \rightarrow \{0,1\}$, represent the type of each gate of $\mathcal{C}$, where s is the I/O size and N the number of gates. In particular, if the i-th gate of $\mathcal{C}$ is an addition gate then $add[i] = 1$, $mul[i] = 0$ and vice versa. We also assume that $add[i] = mul[i] = 0$ for all $i \leq s$. Next, the permutation function $\sigma : [3(N+s)] \rightarrow [3(N+s)]$ describes the wiring of $\mathcal{C}$. In more detail, let $J \in [3N]^k$ be a set of indexes of wires that originate from the same gate and J' a permutation of J. Then σ is defined as $\sigma[J[i]] = J'[i]\ \forall i \in [k]$, and for every set J. To ensure the correct computation of $\mathcal{C}$, the prover interacts with the verifier to prove the that (1) each gate of $\mathcal{C}$ has been computed correctly (known as gate consistency check), and (2) every wire with the same fan-out gate has the same value (known as wiring consistency check). Both (1) and (2) are proven using the sumcheck protocol. Due to space limitations, we provide more details in the full version [37].

2.3 Zero-knowledge Arguments of Knowledge

Argument systems [1] are similar to interactive proofs but focus on bounded adversaries. More precisely, an argument system for an NP language $\mathcal{L}$ enables a PPT prover holding an instance $\mathbf{x}$ and its corresponding witness $\mathbf{w}$ to convince a verifier that $\mathbf{x} \in \mathcal{L}$. We say that an argument system is an argument of knowledge if for every PPT adversary $\mathcal{A}$ who generates a convincing proof, there exists a PPT extractor $\mathcal{E}$ with access to the internal state of $\mathcal{A}$ and extracts a witness $\mathbf{w}$ such that $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}_\mathcal{L}$ with overwhelming probability. An argument system is zero-knowledge if the verifier learns nothing else, other than the fact that $\mathbf{x} \in \mathcal{L}$. Furthermore, if the scheme requires only one round of interaction, then we say that the argument system is a non-interactive argument. In addition, when the proof size and verification time are sub-linear to the size of the witness, then the argument system is succinct. If all properties hold simultaneously, then the argument system is a zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) [1]. Finally, if both the time and space complexities of the prover are preserved with respect to natively certifying $\mathcal{R}_\mathcal{L}$, then the underlying zkSNARK is space-efficient. In our work, we adapt the definition of space-efficient zkSNARKs from [57] for arithmetic circuits. In particular, for a given arithmetic circuit $\mathcal{C}$:

Definition 1 (Space-Efficient zkSNARKs)**.** *A zkSNARK for the relation $\mathcal{R}_\mathcal{C} = \{(\boldsymbol{x}; \boldsymbol{w}) : \mathcal{C}(\boldsymbol{x}, \boldsymbol{w}) = 1\}$ is space-efficient if it satisfies the following performance requirements: (1) The prover runs in time $\tilde{O}(|\mathcal{C}|)$. (2) The prover uses space $O(S_{eval} \cdot poly \log |C| + |\mathcal{C}|^\varepsilon)$, where S_{eval} is the optimal space needed to evaluate $\mathcal{C}$ and $\varepsilon < 1$ a small positive constant.*

Streaming Setting [17]. Space-efficient zkSNARKs are constructed similarly to "standard" zkSNARKs but the prover works in the streaming setting. In particular, the prover only has (read-only) streaming access to the data needed to generate the proof via *streaming oracles* and utilizes only small working buffer space (typically sub-linear to the $|\mathcal{C}|$) to produce a proof. The vast majority of works on space-efficient arguments[1], including ours, treat streaming oracles as routines

[1] To the best of our knowledge, only [19, 20] consider a slightly different streaming setting as discussed previously.

that internally maintain a state and, upon invocation by the prover, use it to compute the next element of the stream. Naturally, in this setting, the total space complexity of the prover consists of the total space needed to run the streaming oracles and the working buffer space used to produce the proof.

Constructing zkSNARKs. Using a PCS, we can construct a zkSNARK from HyperPlonk by replacing the polynomial oracles with commitments [10, 12, 23]. Assuming a linear-time PC scheme with poly-logarithmic proof size and verification time, then the resulting zkSNARK has $O(N+s)$ prover time, $O(s+poly\log N)$ verification time, and $O(poly\log N)$ proof size. Unfortunately, this zkSNARK is not space-efficient since its space complexity is $O(N+s)$. Looking ahead, in Section 5 we show how to modify it in order to make it space-efficient.

3 Space-Efficient Sumcheck with Optimal Prover Time

In this section, we show how to construct a space-efficient sumcheck protocol that achieves $O(N)$ prover time, uses $O(B)$ working buffer space and has proof size and verification time of $O(N/B+\log N)$, where $B \in [\sqrt{N}, N]$. In more detail, we will construct a protocol for proving the instance $K = \sum_{x\in\{0,1\}^n} f(\mathbf{x})g(\mathbf{x})$, where $N = 2^n$ and $f, g : \mathbb{F}^n \to \mathbb{F}$, are the multi-linear extensions of the vectors $\mathbf{A}_f, \mathbf{A}_g \in \mathbb{F}^N$. Our sumcheck works in the streaming setting [17, 19]. Namely, the prover only has streaming access to $\mathbf{A}_f, \mathbf{A}_g$ and maintains a (small) working buffer space used to generate the proof. To ease presentation, we implicitly assume the existence of streaming oracles $S(\mathbf{A}_f), S(\mathbf{A}_g)$ providing access to the coefficients $\mathbf{A}_f$ and $\mathbf{A}_g$. Looking ahead, in Section 5 we will show how to construct them in practice. Furthermore, we will present our protocol as a PIOP, namely, we assume that the verifier has access to the polynomial oracles of f, g. Later, these oracles will be replaced with our space-efficient PCS.

High-Level Idea of our Protocol. As a starting point, a seemingly straightforward solution, would be to partition f, g in N/B polynomials (with B coefficients each) $f_i, g_i : \mathbb{F}^{\log B} \to \mathbb{F}$, such that $f(\mathbf{x}_1, \mathbf{x}_2) = \sum_{\mathbf{i}\in\{0,1\}^{\log N/B}} \beta(\mathbf{i}, \mathbf{x}_1) f_i(\mathbf{x}_2)$ (likewise we define $g(\mathbf{x}_1, \mathbf{x}_2)$). Then, generate N/B sumcheck proofs for $K_i = \sum_{\mathbf{x}\in\{0,1\}^{\log B}} f_i(\mathbf{x})g_i(\mathbf{x})$. The verifier will check if $K = \sum_{i\in[N/B]} K_i$ and validate the correctness of the proofs. Unfortunately, this protocol does not fit our requirements for two reasons. First, because we prove every sumcheck instance independently, the verifier will receive claimed evaluations of partial polynomials on different evaluation points, whereas, we wish to end up with evaluation claims of f, g at the same evaluation point. Although we can solve this issue by using an aggregation scheme for sumcheck instances [13, 58], known protocols require the prover to invoke the sumcheck protocol for instances that involve f and g, leading to the same problem we try to solve. Furthermore, the verification time and proof size are $O(N\log B/B)$ and not $O(N/B)$.

Instead, our prover sequentially accumulates the N/B sumchecks, using the sumcheck folding scheme till it reaches a single one of size B for which it generates a standard sumcheck proof. Finally, the verifier ends up with evaluation claims of the folded polynomials $\tilde{f}, \tilde{g}$ at a random point $\mathbf{v}$. Unfortunately, it can not validate their correctness, since it only has oracle access to f, g. Instead, it reduces these claims into evaluation claims of f, g. To achieve this, the prover first sends the evaluation claims of the partial polynomials at the same point $\{f_i(\mathbf{v}), g_i(\mathbf{v})\}_{i\in[N/B]}$. Then, the verifier uses them to validate the correctness of $\tilde{f}(\mathbf{v}), \tilde{g}(\mathbf{v})$. Next, it produces the evaluation claims of f, g by picking a random point $\mathbf{u} \in \mathbb{F}^{\log(N/B)}$ and computing $f(\mathbf{u}, \mathbf{v}) = \sum_{\mathbf{i}\in\{0,1\}^{\log(N/B)}} \beta(\mathbf{i}, \mathbf{u}) f_i(\mathbf{v})$ (and likewise $g(\mathbf{u}, \mathbf{v})$). We further reduce the verification time and proof size by delegating this computation to the prover via another sumcheck. Due to space limitations, we provide a detailed description of the interactive version of our protocol in the full version along with a formal theorem. Furthermore, we also prove that it satisfies round-by-round soundness [40], which, as shown in [40, 41] maintains the soundness of its non-interactive version, when using Fiat-Shamir in the ROM.

4 A Concretely Efficient Linear-Time PCS

In this section, we present our transparent polynomial commitment scheme with optimal prover time and poly-logarithmic verification time and proof size. Compared to existing polynomial commitment schemes with similar characteristics [12, 47], our PCS is concretely faster and generates a smaller proof size. Furthermore, it has the following extra property: Given streaming access to an N-sized polynomial and a threshold instance size B, it can commit and generate an evaluation proof in time $O(N)$ using only $O(B)$ working buffer space where $B \in [\sqrt{N}, N/\log N]$, and at the same time, the proof size and verification time are $O(N/B + \log^2 N)$. Here, we will present the interactive variant of our PCS in the "standard" setting, i.e., when $B = N$. In the full version we show how to adapt our PCS in the streaming setting and provide formal proofs.

At a high level, our PCS is based on the same framework used to instantiate all other PC schemes with similar characteristics. Namely, we first use a PC scheme with optimal prover time but proof complexity sub-linear to the polynomial size. Then, we reduce the latter to poly-logarithmic via proof composition. We follow the same approach as Orion+ [13] and BrakingBase [47], which use the Brakedown PC [48] (see Section 2.1 for more details) modified as follows.

First, to commit a N-sized polynomial, the prover organizes its coefficients in a matrix with $\log N$ rows and $N/\log N$ columns. Subsequently, it encodes each row using a linear code with linear-time encoding and computes the Merkle root of the "encoded" matrix. By doing this, in the evaluation phase, the prover has to send the aggregated vectors $\mathbf{F}_1, \mathbf{F}_2$, of size $N/\log N$ and l columns of size $\log N$ with their opening

proofs—where l a constant depending on the code distance.

Initially, this modification seems detrimental because we increased the proof size from $O(\sqrt{N})$ to $O(N/\log N)$. However, that increase solely stems from the fact that the prover has to send the vectors $\mathbf{F}_1, \mathbf{F}_2$ whereas the rest of the proof size is poly-logarithmic to N. Furthermore, the verifier only uses these aggregated vectors to perform lookups at l positions of their encodings. Hence, these checks can be delegated to the prover by having it commit to the aggregated rows and prove the correctness of the lookups via a succinct argument of knowledge.

More generally, given a linear code $Enc : \mathbb{F}^K \to \mathbb{F}^{\delta K}$, and a K-sized vector $\mathbf{m}$, the prover needs to generate a proof for the relationship: $\mathcal{R}' = \{(C_m, \mathbf{y}, I; \mathbf{m}, \mathbf{w}) : C_m = Commit(\mathbf{m}) \wedge \mathbf{y} = \mathcal{C}'_d(\mathbf{m}, I, \mathbf{w})\}$. In our case, $\mathbf{m}$ is $\mathbf{F}_1$ or $\mathbf{F}_2$, $K = N/\log N$, and $I = \{i_1, \ldots, i_l\} \in [\delta K]^l$ is the set of column indexes. Finally, $\mathcal{C}'_d$ is an arithmetic circuit that takes input $\mathbf{m}, I$ and auxiliary information $\mathbf{w}$ and outputs the lookups the verifier must make on the encoding of the message $\mathbf{m}$. More formaly, $\mathcal{C}'_d$ computes and outputs a vector $\mathbf{y} \in \mathbb{F}^l$, such that $\mathbf{y}[j] = \mathbf{E}[i_j]$, $\forall j \in [l]$, where $\mathbf{E} = Enc(\mathbf{m})$.

As mentioned in the introduction, while this approach generates significantly smaller proof sizes compared to the other PC schemes with the same properties [12], its proof composition step, i.e., generating a proof for $\mathcal{R}'$, leads to prohibitively slow proving times. That is because, to achieve optimal commitment and evaluation complexity, the prover has to use a linear code with a linear-time encoding that works over any field. However, the only practical code with these properties is the Spielman code [48, 49] whose encoding algorithm is SNARK "unfriendly". That is, the circuit representing the encoding algorithm has "arbitrary" structure, and must take as input the expander graph of the Spielman code, which is much larger than the message size. Unfortunately, to output $\mathbf{y}$ when using a Spielman code, $\mathcal{C}'_d$ must first evaluate $\mathbf{E}$. Hence, it inherits this overhead, especially considering the "large" message size K (= $N/\log N$). In fact, this exact approach is used in the concurrent work of Nair et al., [47]) which also builds a linear-time PC with polylogaritmic proofs but the resulting prover time is $\times 8$ slower than that of Brakedown.

Based on the above, our main focus becomes minimizing this specific overhead, as we describe next.

4.1 Efficient Composition via Tensor Codes

Following the above discussion, it becomes apparent that to have any hope of efficiently proving $\mathcal{R}'$, we must avoid evaluating a Spielman code for large messages inside $\mathcal{C}'_d$. Our key idea is to replace the underlying linear code with a tensor code. In particular, our tensor code uses two linear codes $Enc_1 : \mathbb{F}^{k_1} \to \mathbb{F}^{n_1}$ and $Enc_2 : \mathbb{F}^{k_2} \to \mathbb{F}^{n_2}$ and encodes $\mathbf{m}$ as follows. First, it parses $\mathbf{m}$ as a matrix $\mathbf{M} \in \mathbb{F}^{k_2 \times k_1}$, where $k_1 k_2 = K$ and then computes $\mathbf{M}' \in \mathbb{F}^{k_2 \times n_1}$ by encoding each row of $\mathbf{M}$ using Enc_1. Finally, it computes $\mathbf{E} \in \mathbb{F}^{n_2 \times n_1}$, by encoding each column of $\mathbf{M}'$ using Enc_2.

Initially, the decision to replace a linear code with a tensor code looks somewhat arbitrary. However, this will enable us to significantly reduce the size of $\mathcal{C}'_d$ by "exploiting" the structure of our tensor code. More precisely, observe that using a tensor code, $\mathcal{C}'_d$ does not have to compute the entire codeword $\mathbf{E}$ to output $\mathbf{y}$. Instead, it only needs to evaluate $\mathbf{M}'$ and then only those columns that contain an index in I to evaluate $\mathbf{y}$. In the worst case, this requires l invocations of Enc_2 inside $\mathcal{C}'_d$, one for each such column. Looking ahead, we will show how to further reduce the number of invocations to only one via batching. What remains is to show how to efficiently evaluate $\mathbf{M}'$ inside $\mathcal{C}'_d$. For this, observe that we do not have to use the "expensive" Spielman code to encode each row of $\mathbf{M}$ but choose Enc_1 to be the Reed-Solomon (RS) code, which is "SNARK-friendly". In fact, proving its correct encoding is asymptotically faster than evaluating it, using the techniques of [4]. Since the RS code does not have a linear-time encoding algorithm, we set k_1 to be constant (independent of K). In this way, our tensor code maintains linear-time encoding as long as Enc_2 has a linear-time encoding algorithm.

We are now ready to formally describe our protocol for proving $\mathcal{R}'$. For our tensor code, let $Enc_1 : \mathbb{F}^k \to \mathbb{F}^{2k}$ be a RS code, for a constant k, and $Enc_2 : \mathbb{F}^{K/k} \to \mathbb{F}^{2N/k}$ a Spielman code. First, we describe the structure of the circuit $\mathcal{C}'_d$. Then, we show how to efficiently prove $\mathcal{R}'$.

Description of our $\mathcal{C}'_d$. To begin with, our circuit takes as input the message $\mathbf{m}$ encoded as a matrix $\mathbf{M} \in \mathbb{F}^{K/k \times k}$, and a "sub-matrix" of its tensor code denoted with $\tilde{\mathbf{E}} \in \mathbb{F}^{\Omega(l) \times 2k}$, which consists of the rows of $\mathbf{E} \in \mathbb{F}^{2K/k \times 2k}$, that contain at least one index in I. Then, we partition $\mathcal{C}'_d$ in two sub-circuits $\mathcal{C}'^{(1)}_d$ and $\mathcal{C}'^{(2)}_d$. The first sub-circuit $\mathcal{C}'^{(1)}_d$ takes as input $\mathbf{M}$ and outputs $\mathbf{M}'$ by performing an FFT on each row of $\mathbf{M}$. Next, $\mathcal{C}^{(2)}_d$ takes as input $\mathbf{M}'$ and outputs $\mathbf{y}$. Naively, this can be done by having $\mathcal{C}'^{(2)}_d$ select the columns of $\mathbf{M}$ that contain at least one index in I, compute their Spielman codewords and output their values at the corresponding index. This requires computing at most l Spielman codewords.

We observe that it is possible to further reduce the number of encodings to only one via *batching*, by modifying $\mathcal{C}'^{(2)}_d$ as follows. First, $\mathcal{C}'^{(2)}_d$ takes as additional input a random point $r \in \mathbb{F}$ provided by the verifier and uses it to aggregate the columns of $\mathbf{M}'$ by computing $\mathbf{m}' = \sum_{i \in [2k]} r^i \mathbf{M}'[:i]$. Then, it computes the Spielman codeword of $\mathbf{m}'$, denoted with $\mathbf{e}' \in \mathbb{F}^{2N/k}$. At this point, it is impossible to compute $\mathbf{y}$ from $\mathbf{e}'$. Thankfully, the matrix $\tilde{\mathbf{E}}$ contains $\mathbf{y}$ but $\mathcal{C}'^{(2)}_d$ can not use it directly to output $\mathbf{y}$ since it has no clue that $\tilde{\mathbf{E}}$ is "well-formed", namely, whether it is the sub-matrix of the tensor code of $\mathbf{M}$. To ensure its well-formedness, $\mathcal{C}'^{(2)}_d$ checks if the aggregation of each row of $\tilde{\mathbf{E}}$ (with respect to r), is equal to the corresponding index in $\mathbf{e}'$. This is done, by first parsing each index $i_j \in I$, as a pair $(i_{j,1}, i_{j,2}) \in \mathbb{N} \times \mathbb{N}$ such that $i_j = 2ki_{j,1} + i_{j,2}$, and

checking if $\mathbf{e}'[i_{j,1}] = \sum_{n\in[2k]} r^n \tilde{\mathbf{E}}[i_{j,1}, n]$. Having established the correctness of $\tilde{\mathbf{E}}$, $\mathcal{C}_d'^{(2)}$ outputs $\tilde{\mathbf{E}}[i_{j,1}, i_{j,2}]$ for all $j \in [l]$.

Proving $\mathcal{R}'$. What remains is to show how to generate a proof for $\mathcal{R}'$. To achieve this, we use the argument of knowledge proposed in [59], which combines specialized sumcheck protocols with a PCS for multi-linear polynomials. In our case, this PCS must be plausibly post-quantum, transparent, and have poly-logarithmic proof size and verification complexity. However, it does not need to have optimal prover time. PC schemes with these characteristics already exist [14, 60], and for our construction, we will use the recently proposed WHIR [60] due to its concretely small proof size and verification time. With that in mind, the prover computes $\tilde{\mathbf{E}}$, commits its multi-linear extension using that PCS, and sends it to the verifier. Then, it receives the random point $r \in \mathbb{F}$ from the verifier and uses the PCS to commit some auxiliary information $\mathbf{w}$, which will be needed to prove the correct computation of $\mathbf{e}'$. Next, both parties interact to prove the correct computation of $\mathcal{C}_d'$. This is done by translating each sub-circuit of $\mathcal{C}_d'$ into a sumcheck instance and proving it using the sumcheck[2].

First, to prove $\mathcal{C}_d'^{(1)}$, we use the specialized sumcheck protocol for batch-proving FFT computations of [4]. As for $\mathcal{C}_d'^{(2)}$, observe that all of its checks can be naturally translated into sumcheck instances except that of computing $\mathbf{e}'$. For this, instead of proving the correct computation of the Spielman encoding algorithm, $\mathcal{C}_d'^{(2)}$ takes as input the parity matrix of the Spielman code $\mathbf{H} \in \mathbb{F}^{2K/k \times 2K/k}$ and $\mathbf{e}'$ and certifies the correctness of the codeword by showing that $\mathbf{H} \cdot \mathbf{e}' = \mathbf{0}$. We can prove the latter using a specialized sumcheck protocol for matrix-to-vector multiplication [63]. To maintain a succinct verification time, the prover has to commit and generate an evaluation proof for the multi-linear extension of $\mathbf{H}$. Since it is a sparse matrix with $O(K/k)$ non-zero elements, we use the protocol for sparse polynomial evaluation as presented in [11]. Because the prover can compute the commitment of $\mathbf{H}$ in a pre-processing phase, $\mathbf{w}$ will only consist of the data needed to generate the evaluation proof of $\mathbf{H}$ and $\mathbf{e}'$ (excluding the first N/k elements which are $\mathbf{m}'$).[3] Eventually, all parties end up with random evaluations at the multi-linear extensions of $\mathbf{m}$, $\tilde{\mathbf{E}}$ and $\mathbf{w}$ that can be validated with PC proofs.

4.2 Our PC scheme

Now, we have everything we need to present our PCS (**Construction 2**). The generation algorithm takes as input the size of the polynomial N and a constant k and (1) generates the public parameters of the PC scheme used for the proof composition step by invoking $\mathbf{pp}_{pc} \leftarrow PC.Gen(N/\log N)$, and (2) computes the expander graph of the Spielman code for a message of size $N/(k \log N)$ and $C_{\mathbf{H}}$, the commitment to its parity matrix using the PC scheme and the techniques of [11]. Finally, it outputs $\mathbf{vk} = \{\mathbf{vk}_{pc}, C_{\mathbf{H}}\}$ and $\mathbf{pk} = \{\mathbf{pk}_{pc}, \mathbf{H}\}$.

Given the public parameters, the prover commits to the multi-linear polynomial $f : \mathbb{F}^{\log N} \rightarrow \mathbb{F}$ by first organizing its coefficients in a matrix $\mathbf{A}_f$ with $\log N$ rows and $N/\log N$ columns and computing the encoded matrix $\mathbf{A}_f'$ by encoding every row of $\mathbf{A}_f$ using the tensor code described above. Next, it hashes each column of $\mathbf{A}_f'$ and sets C, the commitment of f, to be the root of the Merkle tree of column hashes.

Subsequently, to prove that $y = f(\mathbf{r})$ the prover works in the following way. First, it receives a random vector $\mathbf{a} \in \mathbb{F}^{\log N}$ from the verifier needed for proximity testing and computes the aggregated vectors $\mathbf{F}_1 = \sum_{i\in[\log N]} \beta(\mathbf{i}, \mathbf{r}_1)\mathbf{A}_f[i:]$, where $\mathbf{r}_1$ consists of the first $\log\log N$ variables of $\mathbf{r}$ and $\mathbf{F}_2 = \sum_{i\in[\log N]} \mathbf{a}[i]\mathbf{A}_f[i:]$. Next, it commits to the multi-linear extension of the concatenation of $\mathbf{F}_1, \mathbf{F}_2$ using the underlying PC scheme and sends its commitment, denoted with C_f, to the verifier. The latter samples the set of column indexes I and sends it to the prover, which replies with the matrix $\mathbf{R}$ containing the columns of $\mathbf{A}_f'$ in I and their opening proofs.

After validating the column membership the verifier has to check that (1) $y = f_{\mathbf{F}_1}(\mathbf{r}_1)$, where $f_{\mathbf{F}_1}$ the multi-linear extension of $\mathbf{F}_1$ and $\mathbf{r}_2$ the last $\log N/\log N$ values of $\mathbf{r}$ and (2) the restriction of the codewords of $\mathbf{F}_1$ and $\mathbf{F}_2$ at I, agree with the inner products of the rows of $\mathbf{R}$ at $(\beta(1, \mathbf{r}_1), \ldots, \beta(\log N, \mathbf{r}_1)$ and $\mathbf{a}$, respectively. For (1), the prover generates an evaluation proof for C_f at the point $(0, \mathbf{r}_2)$. Note that (2) corresponds to the generation of two proofs for $\mathcal{R}'$ using $\mathbf{F}_1$ and $\mathbf{F}_2$, respectively. As an optimization, we batch these two tasks into one by having the verifier sample a random point $b \in \mathbb{F}$ and send it to the prover. Then, the prover generates a single proof for $\mathcal{R}'$ over the aggregated message $\mathbf{F}_1 + b\mathbf{F}_2$ (note that we slightly modify $\mathcal{C}_d'$ to first compute the aggregated message). Upon receiving that proof, the verifier validates its correctness, and checks if for the output $\mathbf{y}$ of $\mathcal{C}_d'$ holds that $\mathbf{y}[j] = \sum_{j\in[\log N]}(\beta(\mathbf{j}, \mathbf{r}_1) + a^j b)\mathbf{R}[i, j]$.

Recall that in the full version we prove knowledge soundness of the interactive version of our PCS. Furthermore, using the same strategy with related work [39, 60, 64] we also show that our PCS satisfies round-by-round soundness.

5 HOBBIT: Space-efficient zkSNARK with Optimal Prover Time

In this section, we will design a space-efficient zkSNARK for any arithmetic circuit $\mathcal{C}$, which given an evaluation algorithm that optimally computes $\mathcal{C}$ in space S_{eval}, achieves prover space complexity of $O(B + S_{eval})$, where $B \in [\sqrt{|\mathcal{C}|}, |\mathcal{C}|/\log|\mathcal{C}|]$ the threshold instance. First, recall

[2]This can be viewed as a specialized variant of the GKR protocol [61] where we use specific customized sumchecks for the computation of each circuit layer. We note these computations are represented by circuits with regular and well-known structures. Thus, the recent GKR vulnerability proposed by Khovratovich et al. [62] does not seem to apply in our case (see [62], Remark 3).

[3]We note that this technique for proving the correctness of a Spielman code is also discussed in concurrent, independent work [47].

Construction 2. *Let $Enc(\cdot): \mathbb{F}^n \to \mathbb{F}^{kn}$ be a linear error-correction code where $k \in \mathbb{N}$, $PC = (Gen, Commit, Eval)$ be a PC scheme, λ the security parameter and $\mathcal{H}(\cdot): \mathbb{F}^* \to \mathbb{F}$ a random oracle. We construct our PC scheme as follows:*

- $\boldsymbol{pk}, \boldsymbol{vk} \leftarrow Gen(N, \lambda)$*: Set $l \leftarrow \mu(\lambda)$, $k \leftarrow \kappa(\lambda)$ and $\boldsymbol{pp}_{PC} \leftarrow PC.Gen(\lambda, kN/\log N)$. Return $\boldsymbol{pp} \leftarrow (l, k, \boldsymbol{pp}_{PC})$.*
- $C \leftarrow Commit(\boldsymbol{pk}, f)$*: Output the commitment C of the multi-linear polynomial $f: \mathbb{F}^{\log N} \to \mathbb{F}$.*
  1. *Arrange the coefficients of f in a matrix $\boldsymbol{A}_f \in \mathbb{F}^{\log N \times (N/\log N)}$ and compute the encoded matrix $\boldsymbol{A}'_f \in \mathbb{F}^{\log N \times (kN/\log N)}$ such that $\boldsymbol{A}'_f[i:] = Enc(\boldsymbol{A}_f[i:])$, for each $i \in [\log N]$.*
  2. *Let $\boldsymbol{h} \in \boldsymbol{F}^{kN/\log N}$ be a vector such that $\boldsymbol{h}[i] = \mathcal{H}(\boldsymbol{A}'_f[:i])$. Set $C \leftarrow MT.Commit(\boldsymbol{h})$.*
- $\Big\langle Eval(\boldsymbol{pk}, f), Verify(\boldsymbol{vk}, C) \Big\rangle (y, \boldsymbol{r})$*: Prove that $f(\boldsymbol{r}) = y$ where $\boldsymbol{r} = (\boldsymbol{r}_1, \boldsymbol{r}_2) \in \mathbb{F}^{\log\log N} \times \mathbb{F}^{\log(N/\log N)}$.*

1. *$\mathcal{V}$: Select a random vector $\boldsymbol{a} \in \mathbb{F}^{\log N}$ and send it to $\mathcal{P}$.*
2. *$\mathcal{P}$: Compute the aggregated rows $\boldsymbol{F}_1 \leftarrow \sum_{i \in [\log N]} \beta(\boldsymbol{i}, \boldsymbol{r}_1) \boldsymbol{A}_f[i:]$ and $\boldsymbol{F}_2 \leftarrow \sum_{i \in [\log N]} \boldsymbol{a}[i] \boldsymbol{A}_f[i:]$. Send C_f, the PC commitment of the multi-linear extension of their concatenation.*
3. *$\mathcal{V}$: Samples a random point $b \in \mathbb{F}$ and the column indexes $I = \{i_1, \ldots, i_l\} \in [kN/\log N]^l$. Sends b, I to $\mathcal{P}$.*
4. *$\mathcal{P}$: Send columns $\boldsymbol{R} \in \mathbb{F}^{l \times \log N}$ such that $\boldsymbol{R}[j:] = \boldsymbol{A}'_f[:i_j]$, for all $i_j \in I$ and an opening proof π_j for each selected column.*
5. *$\mathcal{V}$: Compute the vectors $\boldsymbol{y}, \boldsymbol{h}' \in \mathbb{F}^l$ such that $\boldsymbol{y}[i] = \sum_{j \in [N/B]} (\beta(\boldsymbol{j}, \boldsymbol{r}_1) + \boldsymbol{a}[j] b) \boldsymbol{R}[i, j]$ and $\boldsymbol{h}'[i] = \mathcal{H}(\boldsymbol{R}[i, 0], \ldots, \boldsymbol{R}[i, N/B])$ for all $i \in [l]$. For each $j \in [l]$ check if $1 = MT.Verify(\pi_j, \boldsymbol{h}'[j], i_j)$.*
6. *$\mathcal{P}$-$\mathcal{V}$: Interact following the proof composition protocol as described in Section 4.1 to prove $\mathcal{R}'$ (using the modified $\mathcal{C}'_d$).*
7. *$\mathcal{P}$-$\mathcal{V}$: Interact following the evaluation algorithm of PC to "open" C_f at $(0, \boldsymbol{r}_2)$ showing that $y = f_{\boldsymbol{F}_1}(\boldsymbol{r}_2)$.*

that in order to access all proving data, previous works on space-efficient zkSNARKs either implicitly assumed the existence of streaming oracles establishing access to all necessary data [19, 20] or focused on specific families of circuits [5]. Instead, our goal is to design a zkSNARK for any arithmetic circuit, where the prover accesses all proving data in a streaming fashion by *computing them on the fly*. This introduces two main challenges: (1) Develop a routine that provides streaming access to the proving data. Crucially, in order to achieve our desired asymptotics, we require this routine to use space proportional S_{eval} and (2) Design a zkSNARK that produces a proof given the information provided by this routine.

Addressing the First Challenge. To begin with, let $Eval^{\mathcal{C}}$ be the evaluation algorithm that computes $\mathcal{C}$ in optimal time, i.e., it computes every gate once. Note that every circuit has such an algorithm which at a high level works as follows. First, throughout the computation $Eval^{\mathcal{C}}$ maintains a set of "active" gates for which not all of their parent gates are computed yet. Next, in every step it either (1) computes and inserts to that set a new gate for which all its children are active gates or (2) removes a gate from that set. The order in which the algorithm inserts or removes gates depends on an evaluation strategy hardcoded in $Eval^{\mathcal{C}}$. Assuming that the algorithm halts whenever all gates are removed from the set, it is not hard to see that $Eval^{\mathcal{C}}$ will run for $2N+s$ steps where N is the number of gates and s the circuit's I/O. Note that S_{eval} is the space needed to store the evaluation algorithm plus the maximum number of active gates throughout its computation[4].

Now, to address the first challenge, we construct our routine such that it provides streaming access to its *execution trace* of $Eval^{\mathcal{C}}$, i.e., a record of the sequence of operations it performs. The main reason behind this decision stems from the following observations. First, and as we will show later, an execution trace encompasses all the data the prover needs to generate the proof. Second, and most importantly, we can construct our routine *by solely relying on the evaluation algorithm* allowing it to use space proportional to S_{eval}.

Before describing our routine, we formally define the execution trace $\mathbf{tr}$, which is a $(2N+s)$-sized vector of tuples where $\mathbf{tr}[i]$ stores some information for the i-th computing step. More precisely, if at that step $Eval^{\mathcal{C}}$ computes a new gate then we set $\mathbf{tr}[i]$ to be $(op, \{(v_L, idx_L), (v_R, idx_R), (v_O, idx_O)\})$, where op is the type of the gate (e.g., addition or multiplication gate), idx_L,idx_R,idx_O the labels of the left, right and output gate and v_L, v_R and v_O their values. Otherwise, if $Eval^{\mathcal{C}}$ removes a gate at the i-th step then we set $\mathbf{tr}[i]$ as (v, idx), where idx is the label of the gate being deleted and v its value.

What remains is to show how to instantiate streaming access to $\mathbf{tr}$. For this, we construct a routine, denoted with $S(\mathbf{tr})$, which upon invocation computes "on-the-fly" and outputs the next tuple in $\mathbf{tr}$. In particular, $S(\mathbf{tr})$ works almost identically to $Eval^{\mathcal{C}}$ but with the following difference. First, it maintains a counter ct initially set to zero and stores the input of $Eval^{\mathcal{C}}$ (e.g., the input values of the circuit or auxiliary information needed to generate them). Next, on each invocation, it executes the next computing step of $Eval^{\mathcal{C}}$, outputs $\mathbf{tr}[ct]$, increments ct by one, and waits for the next invocation. Whenever $ct = 2N$, it sets $ct \leftarrow 0$ and repeats the computation from scratch. Observe that the space complexity of $S(\mathbf{tr})$ is $O(S_{eval})$, which derives from the space needed to evaluate the circuit plus some additional space to store its input.

Addressing the Second Challenge. Having established streaming access to $\mathbf{tr}$, we can now focus on constructing HOBBIT. For this, we observe that state-of-the-art schemes

[4] In practice, depending on the circuit structure and evaluation strategy, S_{eval} can be sub-linear to N or even constant [5].

do not directly fit our setting. For instance, schemes that rely on R1CS or Plonk constraint systems require some additional information that depends on the circuit structure and is not directly provided by **tr**. For example, when encoding a circuit using R1CS, the prover must have column-wise access to the R1CS matrices. Likewise, for Plonk, the prover must access the permutation polynomial required to ensure wiring consistency. In fact, prior works that adapted proving systems for these constraint systems in the streaming setting [19, 20] assumed that the prover has streaming access to this information without providing details on how to instantiate them. Unfortunately, as we show with an example in Appendix A, it is unclear how to obtain such information having only streaming access to **tr** without blowing up proving space or time.

Instead, we would like our proof system to directly work over **tr**. Interestingly, proving systems based on the GKR protocol [61] have this property. However, even for these systems a similar problem arises. This is because, to prove the correct computation of each layer in a streaming setting, the prover must access its gates in sequential order. Unfortunately, there is no guarantee that the evaluation algorithm computes the gates in that order. Indeed, prior work [5] on space-efficient GKR focused only on data-parallel circuits, in which they introduced a routine for any such circuit that generates streaming access **tr** in the desired order. Motivated by the above, we construct our zkSNARK by modifying the HyperPlonk [13] proving system to fit our setting. We first show how to instantiate a space-efficient variant of the HyperPlonk PIOP and finally present how to compile it into a zkSNARK using our PCS. We begin by showing how to prove gate and wiring consistency in the streaming setting.

Proving the Correct Computation of Each Gate. Recall from Section 2.2 that in order to ensure the correct computation of each gate, the prover needs to show that $\sum_{\mathbf{i}\in\{0,1\}^{\log(N+s)}} \beta(\mathbf{i},\mathbf{r})(g_{add}(\mathbf{i})(f_L(\mathbf{i}) + f_R(\mathbf{i})) + g_{mul}(\mathbf{i}) f_R(\mathbf{i}) f_L(\mathbf{i}) - f_{\mathbf{O}}(\mathbf{i}) + \mathbf{I}(\mathbf{i}))$ is equal to zero. To prove this, we use our space-efficient sumcheck protocol described in Section 3. What remains is to show how to establish streaming access to the coefficients of these polynomials. To achieve this, for each polynomial, we construct a routine that internally invokes $S(\mathbf{tr})$, parses its reply, and outputs the desired coefficient.

In more detail, the streaming oracle of the coefficients of g_{add} (or g_{mul} resp.), upon invocation, calls $S(\mathbf{tr})$ until it receives a response corresponding to a newly computed gate. Then, returns 1 if the gate is addition (or multiplication resp.) and 0 otherwise. Likewise, we construct the streaming oracles for the coefficients of f_L, f_R, f_O. Namely, on the i-th invocation, each oracle outputs the value of the left/right input and output of the i-th newly computed gate. Next, to construct the streaming oracle of $\beta(\mathbf{i},\mathbf{r})$, we rely on the techniques of [5]. Finally, it is straightforward to instantiate streaming access to the polynomial $I(\mathbf{x})$ as both parties store it locally.

Proving Wiring Consistency. What remains is to show that the output wires of each gate have the same value. The "classic" way of proving this is by showing that $\prod_{\mathbf{i}\in\{0,1\}^{\log 4(N+s)}}(i + af(\mathbf{i})) = \prod_{\mathbf{i}\in\{0,1\}^{\log 4(N+s)}}(f_\sigma(\mathbf{i}) + af(\mathbf{i}))$. To prove the above equation in a space-efficient manner, it is necessary to establish streaming access to the coefficients of f and σ. Unfortunately, although we can directly construct a streaming oracle for the coefficients of f using $S(\mathbf{tr})$, it is not clear how to achieve this for the coefficients of f_σ. That is because, in order to compute $f_\sigma(\mathbf{i})$, the prover must know when another fan-out wire from the same gate will be used throughout the entire computation. As discussed earlier, it is not obvious how to obtain such information efficiently when only using $S(\mathbf{tr})$.

To circumvent that issue, we will convert the task of proving wiring consistency into proving the correctness of accesses from a memory that stores all gate values. In more detail, let $\{(\mathbf{u}[i], \mathbf{idx}[i])\}_{i\in[4(N+s)]}$ be a set of pairs, where $\mathbf{u}$ is the vector of wire values for which f is multi-linear extension and $\mathbf{idx}[i]$ contains the label of the gate this wire originates from. Furthermore, let $\mathbf{v} \in \mathbb{F}^N$ be the vector such that $\mathbf{v}[i]$ corresponds to the value of the gate with label i. By considering $\mathbf{v}$ as a read-only random access memory, observe that wiring consistency holds *if and only if* $\boldsymbol{u}[i] = \boldsymbol{v}[\boldsymbol{idx}[i]],\ \forall i \in [4(N+s)]$. To prove the latter, we will use the offline memory-checking technique of [65] which will enable us to prove wiring consistency by only using streaming access to **tr**.

First, the memory checking technique of [65] considers the following four sets. A set representing the initial and final state of the memory which we define as $\mathcal{H}_I = \{(i, \mathbf{v}[i], 0)\}_{i\in[N+s]}$ and $\mathcal{H}_F = \{(i, \mathbf{v}[i], \mathbf{f}[i])\}_{i\in[N+s]}$ respectively, where $\mathbf{f}[i]$ denotes the total number of times we accessed the gate with label i (i.e., its fan-out). In addition, let $\mathcal{H}_R = \{(\mathbf{idx}[i], \mathbf{u}[i], \mathbf{r}[i])\}_{i\in[4(N+s)]}$, $\mathcal{H}_W = \{(\mathbf{idx}[i], \mathbf{u}[i], \mathbf{wr}[i] = \mathbf{rd}[i]+1)\}_{i\in[4(N+s)]}$ be the read/write sets storing information of each read operation where $\mathbf{rd}[i]$ indicates the number of times we accessed a wire with index $\mathbf{idx}[i]$ until the i-th step. Next, to prove that memory accesses are consistent, we need to show that (1) $\mathbf{wr} = \mathbf{rd} + \mathbf{1}$ and (2) $\mathcal{H}_W \cup \mathcal{H}_I = \mathcal{H}_R \cup \mathcal{H}_F$. Note that since $\mathbf{wr}, \mathbf{rd}$ are independent of the circuit's input, we can prove (1) in an offline pre-processing phase. For (2), we follow the techniques first presented in [11, 66] but adapt them in the streaming setting. To achieve this, we first construct a prover that works in the streaming setting and proves equation (2). Subsequently, we show how to instantiate streaming access to its proving data.

5.1 Proving Memory Consistency in the Streaming Setting

Following the observations of previous work [11, 65], we convert the task of checking (2) to showing that $a_I a_W = a_F a_R$, where:

- $a_I = \prod_{\mathbf{i}\in\{0,1\}^{\log N}}(f_{\mathbf{addr}}(\mathbf{i}) + a f_{\mathbf{v}}(\mathbf{i}))$
- $a_F = \prod_{\mathbf{i}\in\{0,1\}^{\log N}}(f_{\mathbf{addr}}(\mathbf{i}) + a f_{\mathbf{v}}(\mathbf{i}) + b f_{\mathbf{f}}(\mathbf{i}))$

- $a_R = \prod_{\mathbf{i}\in\{0,1\}^{\log N}}(f_{\mathbf{idx}}(\mathbf{i}) + af(\mathbf{i}) + bf_{\mathbf{rd}}(\mathbf{i}))$
- $a_W = \prod_{\mathbf{i}\in\{0,1\}^{\log N}}(f_{\mathbf{idx}}(\mathbf{i}) + af(\mathbf{i}) + bf_{\mathbf{wr}}(\mathbf{i}))$

For random points $a, b \in \mathbb{F}$ and $f_{\mathbf{v}}$, $f_{\mathbf{rd}}$, $f_{\mathbf{wr}}$, $f_{\mathbf{f}}$, $f_{\mathbf{idx}}$ and $f_{\mathbf{addr}}$ the multi-linear extensions of $\mathbf{v}, \mathbf{rd}$, $\mathbf{wr}$, $\mathbf{f}$, $\mathbf{idx}$ and $\mathbf{addr}$ respectively. Consequently, to prove (2), we need to show that the four products have been correctly computed. To simplify the discussion, we will focus on proving the correct computation of a single product and more specifically, we will show how to prove $p = \prod_{i\in[N]}\mathbf{x}[i]$ when the prover has only streaming access to $\mathbf{x}$ (at the end of this section, we will show how to actually instantiate streaming access to $\mathbf{x}$). In this work, we will present two different protocols that offer different performance trade-offs. In particular, the first one produces a smaller proof and has a shorter verification time, while the second one has a concretely faster prover. In our implementation, we use the latter since the additional overhead in proof size is almost negligible in practice.

Our First Protocol. This protocol is a slightly modified version of [20, 67]. To begin with, let $\mathbf{a} \in \mathbb{F}^{N+1}$ be a vector defined as $\mathbf{a}[0] = 1$ and $\mathbf{a}[i] = \prod_{j\in[i]}\mathbf{x}[j]$, for all $i \in \{1,\ldots,N+1\}$ and $f_1, f_2 : \mathbb{F}^{\log N} \to \mathbb{F}$, the multi-linear extensions of $\mathbf{a}_1 = \mathbf{a}[:N]$ and $\mathbf{a}_2 = \mathbf{a}[1:N+1]$ respectively. To prove $p = \prod_{i\in[N]}\mathbf{x}[i]$, we need to ensure that: (i) $\mathbf{x}[i]\mathbf{a}_1[i] = \mathbf{a}_2[i]$ $\forall i \in [N]$, (ii) $\mathbf{a}_1[i+1] = \mathbf{a}_2[i]$, for all $i \in \{1,\ldots,N\}$ and (iii) $\mathbf{a}_2[N-1] = p$ and $\mathbf{a}_1[0] = 1$. For (i), we use our sumcheck protocol for the instance $\sum_{\mathbf{i}\in\{0,1\}^{\log N}}\beta(\mathbf{i},\mathbf{r}_1)(f_1(\mathbf{i})f_{\mathbf{x}}(\mathbf{i}) - f_2(\mathbf{i}))$ where $\mathbf{r}_1 \in \mathbb{F}^{\log N}$ is a random point from the verifier. Similarly with (ii), but will also use the multi-linear polynomial $next(\mathbf{i},\mathbf{j})$ as defined in [68] such that $next(\mathbf{i},\mathbf{j}) = 1$, iff $i = j+1$. Given this, we convert equation (ii) into a sumcheck instance of the form $f_2(\mathbf{r}_2) - \beta(\mathbf{0},\mathbf{r}_1)p = \sum_{\mathbf{i}\in\{0,1\}^{\log N}} next(\mathbf{r}_1,\mathbf{i})f_1(\mathbf{i})$ where $\mathbf{r}_2 \in \mathbb{F}^{\log N}$ is a random point selected by the verifier. For (iii), the prover needs to show that $f_2(\mathbf{N\text{-}1}) = p$ and $f_1(\mathbf{0}) = 1$.

Based on the above, to prove the correctness of p, the prover and verifier interact over the following protocol. First, the prover commits to f_1, f_2 using the space-efficient variant of our PCS and sends the commitments to the verifier. Next, both parties interact following our space-efficient sumcheck protocol over (i),(ii). At the end of this protocol, the verifier holds evaluation claims of f_1, f_2, f and $next$ at random points. Next, the prover interacts once again with the verifier using our space-efficient PCS to prove the validity of the evaluation claims of f_1, f_2. Last but not least, the verifier validates the correctness of the evaluation claim of $next$ locally (which can be done in $O(\log^2 N)$ time [68]) and outputs the evaluation claims of f (which will be validated later in our final protocol).

Instantiating Streaming Access to $\mathbf{a}_1, \mathbf{a}_2$ and the coefficients of next. It remains to show how to establish streaming access to the "auxiliary" proving data. First, $\mathbf{a}_1, \mathbf{a}_2$ can be instantiated directly using $S(\mathbf{x})$ and an additional $O(1)$ space. Furthermore, we can instantiate streaming access to the coefficients of $next(\mathbf{r}_2,\mathbf{i})$ using $O(\sqrt{N})$ space the same way that we produced streaming access $\beta(\mathbf{r},\mathbf{i})$ but shifting the output by one.

Performance Analysis. It is not hard to see that the prover time is $O(N)$, the proof size and verification time are $O(N/B)$ while the working buffer space needed by the prover is $O(B)$.

Our Second Protocol. For this protocol we will the techniques of [63] which translate the task of proving the correctness of the product into proving the correct computation of a circuit with a binary tree structure with only multiplication gates. We adapt the protocol of [63] in the streaming setting, using the space-efficient GKR protocol as described in [5] but replace the sum check of each layer with the specialized instance of the form $\sum_{\mathbf{x}\in\{0,1\}^{\log(N/2^i)}}\beta(\mathbf{i},\mathbf{r}_{i+1})f_L^{(i)}(\mathbf{x})f_R^{(i)}(\mathbf{x})$. Observe that since this circuit has $\log N$ depth, naively running the space-efficient GKR would lead to a $O(N\log N)$ prover time. Instead, as described in [5] we need to reduce its depth. Next, we will show how to efficiently achieve this, specifically for product trees.

To begin with, let d be a small constant that represents the new depth of the circuit. Our strategy is to horizontally partition the circuit into $(\log N)/d$ sub-circuits and then prove their correct execution "in parallel". To achieve this, the prover commits to all layers $i \in [\log N]$ such that $i\%(\log N)/d = 0$ using the space-efficient variant of our PCS. For example, if $d = 5$ and the circuit has 20 layers, then the prover will have to commit the 5th,10th and 15th layer. Note that this can be done in $O(N)$ time and $O(B)$ space by running the commitment algorithm for each layer "in parallel" and also decreasing the working buffer space of our PCS by $B/\log N$. Next, we batch-prove the $(\log N)/d$ sub-circuits using the space-efficient variant of the GKR protocol of [5] but replacing their space-efficient sumcheck protocol with ours. When the GKR protocol completes, the verifier ends up with evaluation claims on the inputs of each sub-circuit (e.g., in the previous example these claims correspond to the original input layer, the 5th,10th and 15th layer). To prove their correctness, the prover runs the evaluation algorithm of our PCS for each layer "in parallel" as it did in the commitment phase.

Performance Analysis. Due the the linear-time complexity of our sumcheck protocol, the GKR protocol requires $O(N)$ time leading to an overall $O(N)$ prover complexity. Regarding proof size, observe that the evaluation space of the product tree is $O(\log|\mathcal{C}|)$, hence the total working buffer space needed is $O(B + \log N) = O(B)$. Regarding proof size, the GKR requires the invocation of $O(d)$ batch sumcheck instances leading to $O(dN/B)$ proof size. Furthermore, at the end of the protocol, the prover has to generate $\log N/d$ opening proofs for polynomials of size $N, N/2^d, \ldots, 2^d$ using a buffer space of $B/\log N$ for each polynomial hence leading to a total proof size of $O((\log N/B)\sum_{i\in[\log N/d]}N/2^{d\cdot i}) = O(N\log N/B)$. Likewise we argue about verification time.

Instantiating Streaming Access to the Proving Data. Observe that to prove the correct computation of the four prod-

ucts a_I, a_W, a_F and a_R, in the streaming setting, we need to establish streaming access to the vectors $\mathbf{v}$, $\mathbf{u}$, $\mathbf{rd}$, $\mathbf{wr}$, $\mathbf{f}$, $\mathbf{idx}$ and $\mathbf{addr}$. Starting from $\mathbf{rd}$, its streaming oracle $S(\mathbf{rd})$ maintains a key-value storage $\mathbf{T}$ (e.g., by using a hash table of $O(S_{eval})$ size), and upon invocation calls $S(\mathbf{tr})$. If its reply corresponds to a newly computed gate (e.g., a tuple of the form $op, (v_L, idx_L), (v_R, idx_R), (v_O, idx_O)$), $S(\mathbf{r})$ outputs ($\mathbf{T}[idx_L]$,$\mathbf{T}[idx_R]$,0) and sets $\mathbf{T}[idx_L] = \mathbf{T}[idx_L] + 1$, $\mathbf{T}[idx_R] = \mathbf{T}[idx_R] + 1$ and $\mathbf{T}[idx_O] = 1$. Otherwise, whenever $S(\mathbf{tr})$ replies with a deletion gate, $S(\mathbf{rd})$ removes idx from $\mathbf{T}$ and recursively invokes $S(\mathbf{tr})$ until receiving a newly computed gate. Likewise, we construct $S(\mathbf{wr})$, but all its outputs are incremented by one. For $\mathbf{idx}$, its streaming oracle does not need to maintain a key-value storage but only output (idx_L, idx_R, idx_O) whenever it gets a newly computed gate from $S(\mathbf{tr})$. We follow a similar logic to construct $S(\mathbf{v}), S(\mathbf{addr})$ and $S(\mathbf{f})$. Namely, for each streaming oracle we use a key-value storage $\mathbf{T}$, and update its contents similarly to $S(\mathbf{r})$. However, they only output elements whenever $S(\mathbf{tr})$ returns a gate deletion reply. In particular $S(\mathbf{v})$, $S(\mathbf{addr})$ and $S(\mathbf{f})$ output v, idx and $\mathbf{T}[idx]$ respectively. Finally, $S(\mathbf{u})$ is constructed in the same way as in the gate consistency check.

5.2 Putting Everything Together

Armed with our basic building blocks, we now have everything we need to construct HOBBIT. Initially, in a preprocessing (witness independent) phase, the prover commits to the representation of the circuit. In particularly, let $f_{\mathbf{c}} : \mathbb{F}^{4+\log(N+s)}$ be the multi-linear extension of the vector $\mathbf{c} = (\mathbf{idx}, \mathbf{rd}, \mathbf{wr}, \mathbf{addr}, \mathbf{f}, \mathbf{add}, \mathbf{mul}, \mathbf{0}) \in \mathbb{F}^{16(N+s)}$ (padded with zeros). The prover commits to $f_{\mathbf{c}}$ using the space-efficient variant of our PCS. Note that it is straightforward to instantiate streaming access to $\mathbf{c}$ using the streaming oracle of each sub-vector. We denote with $C_{\mathbf{c}}$ the commitment of $f_{\mathbf{c}}$.

Given a commitment to $f_{\mathbf{c}}$ the prover can know generate a proof for the relation $\mathcal{R}_{\mathcal{C}} = \{(\mathbf{x}; \mathbf{w}) : \mathcal{C}(\mathbf{x}, \mathbf{w}) = 1\}$ as follows. First, it commits to $f_{\mathbf{w}'} : \mathbb{F}^{3+\log N} \rightarrow \mathbb{F}$, the multi-linear extension of $\mathbf{w}'$ defined as $\mathbf{w}' = (\mathbf{u}, \mathbf{v}) \in \mathbb{F}^{4N}$ using the space-efficient variant of our PCS (note that $\mathbf{w} \subseteq \mathbf{w}'$). Observe that the prover can directly instantiate streaming access to $\mathbf{w}'$ using $S(\mathbf{u})$ and $S(\mathbf{v})$. Then, it proves the correct computation of $\mathcal{C}$, using the protocols described above. Upon completion, the verifier ends up with the evaluation claims y_1, y_2 of $f_{\mathbf{w}}$ at $\mathbf{r}_1, \mathbf{r}_2$ and y_1', y_2' of $f_{\mathbf{c}}$ at $\mathbf{r}_1'$ and $\mathbf{r}_2'$. To reduce these evaluation claims to $(y_3, \mathbf{r}_3)$ and $(y_3', \mathbf{r}_3')$ both parties interact following the aggregation protocol described in [5, 13]. Finally, the prover generates the evaluation proofs of the reduced claims using $C_{\mathbf{w}}$ and $C_{\mathbf{c}}$ respectively.

Finally, we can claim the following which we prove in the full version:

Theorem 1. *Construction 3 is a space-efficient succinct argument of knowledge for the relationship* $\mathcal{R} = \{(\boldsymbol{x}; \boldsymbol{w}) : \mathcal{C}(\boldsymbol{x}, \boldsymbol{w}) = 1\}$ *with* $O(|\mathcal{C}|)$ *prover time,* $O(|\mathcal{C}|/B + \log^2 B)$ *verification time and proof size and* $O(B + S_{eval})$ *space complexity, where* S_{eval} *is the space needed to evaluate* $\mathcal{C}$ *and* $B \in [\sqrt{|\mathcal{C}|}, |\mathcal{C}|]$ *a threshold instance size.*

5.3 Supporting Lookup Arguments

Up until this point, we explained our construction to support only multiplication and addition gates. To efficiently support non-linear operations such as comparisons and bitwise operations and many more, we need to embed a lookup argument in our zkSNARKs. At a very high level, lookup arguments [9, 66] enable a prover to show that all elements of a committed vector $\mathbf{x}$ belong to a predetermined committed vector $\mathbf{t}$ called lookup table. More formally, they enable a prover to generate a proof for the following relationship $\mathcal{R}_{lkp} = \{(C_{\mathbf{x}}, C_{\mathbf{t}}); (\mathbf{x}, \mathbf{t}) : C_{\mathbf{x}} = PC.Commit(\mathbf{pk}, f_{\mathbf{x}}) \wedge C_{\mathbf{t}} = PC.Commit(\mathbf{pk}, f_{\mathbf{t}}) \wedge \mathbf{x} \subseteq \mathbf{t}\}$. In this section, we first show how to construct a lookup argument in the streaming setting, which can be seen as a space-efficient variant of Lasso [66], and then, using the techniques of Plonkup [69], present how to embed it in our zkSNARK.

Our Space-Efficient Lookup Argument. Given streaming access to $\mathbf{x}$, we will construct a space-efficient lookup argument that achieves $O(|\mathbf{x}| + |\mathbf{t}|)$ prover time, $O(|\mathbf{x}|/B + \log^2(|\mathbf{x}| + |\mathbf{t}|))$ verification time and proof size and $O(B + |\mathbf{t}|)$ space complexity. We will rely on the Lasso lookup argument [66] which is based on the offline memory checking technique of [65]. Recall from the previous section that [65] uses four sets $\mathcal{H}_I, \mathcal{H}_F, \mathcal{H}_R, \mathcal{H}_W$ defined as follows: $\mathcal{H}_I = \{(i, \mathbf{t}[i], 0)\}_{i \in [|\mathbf{t}|]}$, $\mathcal{H}_I = \{(i, \mathbf{t}[i], \mathbf{f}[i])\}_{i \in [|\mathbf{t}|]}$, $\mathcal{H}_R = \{(i, \mathbf{x}[i], \mathbf{r}[i])\}$ and $\mathcal{H}_W = \{(i, \mathbf{x}[i], \mathbf{w}[i] = \mathbf{r}[i] + 1)\}$. The prover has to show that: (1) $\mathbf{w} = \mathbf{r} + 1$ and (2) $\mathcal{H}_I \cup \mathcal{H}_W = \mathcal{H}_F \cup \mathcal{H}_R$.

In our protocol, the prover first commits to $\mathbf{r}, \mathbf{w}$ using the space-efficient variant of our PCS and $\mathbf{f}$ using the standard our PCS and sends all commitments to the verifier. Now, to prove (1), the verifier picks a random point $\mathbf{r}_1$, sends it to the prover which replies with the evaluation claims of y_w, y_r. Next, the verifier checks if $y_w = y_r + 1$ and interacts with the prover to validate the correctness of the claims. For (2), both parties follow the identical protocol with the one used in the wiring consistency test.

It remains to show how to instantiate streaming access to $\mathbf{r}, \mathbf{w}$. For $\mathbf{r}$ we construct a routine that maintains a key-value storage $\mathbf{T}$ of size $|\mathbf{t}|$ initialized to $\{0\}^{|\mathbf{t}|}$. Upon the i-th invocation, it calls the streaming oracle of $\mathbf{x}$, receives $\mathbf{x}[i]$ outputs $\mathbf{T}[\mathbf{x}[i]]$ and sets $\mathbf{T}[\mathbf{x}[i]] = \mathbf{T}[\mathbf{x}[i]] + 1$. Likewise, we instantiate streaming access to $\mathbf{w}$ but each output is incremented by one.

Embedding our Lookup Argument into our zkSNARK. To embed our lookup argument into our zkSNARK we will use the same techniques proposed by Plonkup [69]. First, we add an additional selector function denoted with $g_{lkp} : [N + s] \rightarrow \{0, 1\}$. Furthermore, just before invoking the space-efficient sumcheck for the gate consistency check, the prover

receives a random point $s \in \mathbb{F}$ from the verifier and commits to $f_{\mathbf{x}}$ defined as $f_{\mathbf{x}}(\mathbf{i}) = sf_L(\mathbf{i}) + s^2 f_R(\mathbf{i}) - f_O(\mathbf{i})$, if the i-gate corresponds to a lookup gate. Otherwise, $f_{\mathbf{x}}(\mathbf{i})$ is set to a default value. Note that we can directly instantiate streaming access to the coefficients of $f_{\mathbf{x}}$ using $S(\mathbf{tr})$. Then, we modify our gate consistency sumcheck instance as follows:

$$\sum_{\mathbf{i}\in\{0,1\}^{\log(N+s)}} \beta(\mathbf{i},\mathbf{r})\big(f_{add_L}(\mathbf{i})f_L(\mathbf{i}) + f_{add_R}(\mathbf{i})f_R(\mathbf{i}) + f_{mul}(\mathbf{i})f_L(\mathbf{i})f_R(\mathbf{i}) + f_{lkp}(\mathbf{i})(sf_L(\mathbf{i}) + s^2 f_R(\mathbf{i}) - f_{\mathbf{x}}(\mathbf{i})) - f_O(\mathbf{i}) + I(\mathbf{i})\big)$$

Finally, by treating $sf_L(\mathbf{i}) + s^2 f_R(\mathbf{i})$ as the index of the lookup table, we invoke our lookup argument as described in the previous paragraph. Observe that the wiring consistency check remains unchanged.

Asymptotically, our zkSNARK with lookup arguments has the same prover complexity and space complexity of $O(B + S_{eval} + |T|)$, where $|T|$ is the space occupied by the lookup tables. Note that for most basic operations, these lookup tables have a small constant size [70].

6 Experimental Evaluation

We implemented and experimentally evaluated our PCS (its standard and space-efficient variants) and HOBBIT. In this section, we present our results.

Implementation Details. Our constructions are implemented in C++ with 8000 lines of code. For field operations, we used the 61-bit Mersenne prime field extension of [14], and for hashes we used the Blake3 hash function [71].

Implementation Details of our PCS. We used the recently proposed WHIR [60] as the underlying PCS. Note that WHIR requires quasi-linear working space to the polynomial size. That would increase the working buffer space of the space-efficient variant of our PCS to $O(B \log B)$. To ensure $O(B)$ space complexity, we modify WHIR relying on the techniques of [14]. Namely, we segment the polynomial in logarithmically many sub-polynomials and later aggregate them. Finally, to instantiate our tensor code, we set $k = 2^{13}$, and for Enc_2 we use the Spielman code implementation of [12] and configure it to have a relative distance equal to $\delta_2 = 0.07$. By setting $\delta_1 = 0.5$, our tensorcode has distance of $\delta = 0.035$ [51].

Implementation Details of our zkSNARK. Our implementation provides an API for circuit evaluation similar to the one of Plonky2 [72]. In more detail, our API consists of addition/multiplication gates, deletion gates, lookups such as `Xor`, range, and less-than gates. Lookup gates are not hardcoded to our zkSNARK, and the user can easily create a new lookup gate by just initializing its lookup table.

*Implementation of S(**tr**).* Initially, the user provides the circuit evaluation algorithm using the API described above. Then, the prover *automatically* instantiates the streaming oracle. First, it creates a new thread using the `std::thread` library of the evaluation algorithm. Both threads "communicate" over a

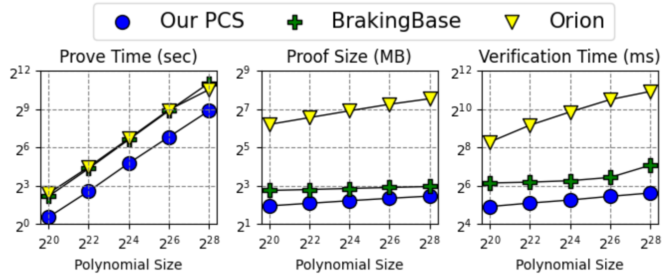

Figure 1: Total (commit plus evaluate) Prover time (left), proof size (middle) and verification time (right) of our PCS, BrakingBase, and Orion for polynomial size 2^{20}-2^{28}.

shared array, and the evaluation thread is initially blocked. When the prover invokes the oracle, it unblocks the evaluation thread and waits. The evaluation thread computes new gates or deletes existing ones. Each gate function performs the desired operation but also writes to the next position of the shared array reply. In addition, if it writes the last array element, it blocks and unblocks the prover thread. To improve performance, we set the shared array size proportional to B.

Experimental Setup. We run our benchmarks using a Linux Ubuntu 20.04.6 LTS with 131GB of RAM and Intel(R) Xeon(R) E-2174G CPU, with 8 cores at 3.80GHz. In all experiments, we used a single thread and only utilized memory. To measure space, we used a script that reports the `VmRSS` field of the `/proc/[pid]/status/` Linux file once per second.

6.1 Benchmarking our PCS

First, we test the performance of our PCS. In the standard setting (i.e., when the prover uses space proportional to the polynomial size), we compare it with the existing linear-time PC schemes with $O(polylogN)$ proof complexity for polynomial size ranging from 2^{20}-2^{28}. In the streaming setting, we test our PCS for (1) constant threshold instance $B = 2^{20}$ and polynomial size 2^{24}-2^{27} and (2) a constant polynomial size $N = 2^{26}$ and threshold instance B between 2^{18}-2^{21}.

Benchmarking our PCS in the Standard Setting. We compare the total prover time (i.e., commit plus evaluation), proof size and verification time of our PCS with the existing linear-time PCS that have poly-logarithmic proof size Orion [12][5], BrakingBase [47]. Note that all the above schemes use Spielman as the underlying linear code. Similarly to our PCS, we select the configuration of [73] that achieves a relative distance of 0.07. For fairness, all PC schemes work in the same field. Furthermore, we replaced the BaseFold [39] and Virgo [14] PCS used in the proof composition phase of BrakingBase and Orion with the state-of-the-art WHIR PCS [60]. In this way, we significantly reduce the proof size of both

[5]There is a discrepancy between the pseudocode and its implementation/evaluation. In this work, we use Orion as described in the former, since the latter has a linear verification time. See Appendix B for more details.

	Commit (sec)		Eval (sec)		Verify (ms)		$|\pi|$ (MB)	
N=2^{24}	23.9	25.7	10.7	9.8	32	8.8	3.3	0.64
N=2^{25}	47.2	50.5	19.8	17.7	38	9.5	4.8	0.8
N=2^{26}	93.6	100.4	37.5	33.7	49	10.8	7.7	1.2
N=2^{27}	186.1	199.6	73.1	65.5	72	13.5	13.6	1.8

	Commit (sec)		Eval (sec)		Verify (ms)		$|\pi|$ (MB)	
B=2^{18}	89.5	92.7	33.9	31.8	111	17	25	3.2
B=2^{19}	91.3	96.8	34.7	32.4	69	12	13.4	1.8
B=2^{20}	93.6	100.4	37.5	33.7	49	10.8	7.7	1.1
B=2^{21}	101.1	112.4	38.5	38.6	41	10	5	0.85

Table 2: Performance of the SL-based (☐ cells) and RS-based (■ cells) variant for (1) variable $N = 2^{24}$-2^{27} and constant $B = 2^{20}$ (which translates to a working buffer space of 472.4MB) and (2) variable $B = 2^{18}$-2^{21} for constant $N = 2^{26}$.

schemes while maintaining the same prover time for BrakingBase and significantly faster times for Orion. Last but not least, we set the number of queries l to ensure 100bits of security. Following the analysis of [48, 53], we set $l = 5,906$ for our PC, $l = 2,935$ BrakingBase, and $l = 42,402$ for Orion[6].

Figure 1 shows the prover time (left), proof size (middle), and verification time (right) of our PC, BrakingBase and Orion for polynomials of varying size. First, observe that our PCS outperforms all other schemes in prover/verification time and proof size. In more detail, it is $\times$3.1-4.5 faster than BrakingBase and $\times$3.16-3.6 faster than Orion. For instance, for $N = 2^{26}$ variables, our PCS takes 112.27sec while BrakingBase and Orion take 465.48sec and 487.28sec respectively. Interestingly, while the commitment time of BrakingBase is $\times$2.5 faster than ours, its evaluation time is $\times$16 slower! As discussed in Section 2.1, this stems from the overhead incurred by proving Spielman encoding for a large message. Furthermore, our PCS produces $\times$1.41-1.74 and $\times$19-34 smaller proofs compared to BrakingBase and Orion. In particular, for $N = 2^{26}$, the proof size of our PCS is 5MB compared to the 7.4MB of BrakingBase and 152.1MB of Orion. A similar trend appears with verification time. Our scheme is $\times$2.3-3 and $\times$10-38 faster than BrakingBase and Orion.

Finally, we also compare our scheme with Brakedown—the state-of-the-art PCS with extremely fast prover time but sublinear proof size and verification time. As expected our PCS achieves $\times$1.03-8.26 smaller proofs and $\times$8.7-29.7 faster verifier. What is more important, however, all these benefits come with a cost of only $\times$2.6 overhead in prover time!

Benchmarking our PCS in the Streaming Setting. For this benchmark, we use two instantiations of our PCS. The first one, which we call "SL-based", uses the tensorcode that we described in the beginning of this section. The second one, which we call "RS-based" uses the RS code to encode both rows and columns. Asymptotically, both schemes have the same proof size and verification time but the RS-based suffers from a $\log B$ multiplicative overhead in prover time. However, it requires significantly fewer queries for the same security level (e.g., $l = 796$ for 100bits of security).

Table 2 (left) shows the performance of both schemes when using a constant threshold instance $B = 2^{20}$ and increasing polynomial size. As expected, the commitment time of the SL-based scheme is faster than the RS-based. Interestingly, we observe the opposite for the evaluation time. That is because when computing the reply matrix $\mathbf{R}$ (see Step 4 of **Construction 2**), the RS-based has to encode to fewer columns since its l is smaller. Finally, due to the larger distance of the RS-based PCS, the proof size and verification time are smaller than the SL-based. Regarding space utilization, the working buffer space used by the prover is the same for both variants and equals to 472.4MB the majority of which is used to store the Merkle Tree.

Impact of B. Table 2 (right) reports the performance of both variants on varying threshold instance size and constant polynomial size. Naturally, the working buffer space scales almost linearly increasing from 150.87MB to 1.8GB. As for prover time, it slightly increases from 123.25-139.6sec for SL-based and 124.5-151sec for the RS-based. This is mainly due to the fact that the instance size of the proof composition increases and hence the evaluation time. At the same time, both the verification time and proof size decrease which aligns with the flexibility property of HOBBIT as discussed in the introduction. Last but not least, we note that the more we increase B, the faster the SL-based prover gets compared to the RS-based.

6.2 HOBBIT Benchmarks

In this section, we test the performance of HOBBIT when proving: (a) the computation of a *layered arbitrary log-space uniform arithmetic circuit* consisting of four equal-sized layers and a total number of gates ranging between 2^{24}-2^{28}, (b) the *inference of a pruned multi-layer perceptron (MLP)* [4, 6] having 20%-60% active weights and and original size 17million weights, (c) the *batch computation of multiple AES128 instances* ranging from $2^{11} - 2^{15}$ [75], and (d) the *correct execution of a SQL query* [76] that selects elements lying in a specific range and outputs the element with the maximum value from a database with rows ranging from 2^{18}-2^{22}. In all experiments, we use the RS-based variant of our PCS. This results in a slightly slower prover time, but smaller proofs.

Circuit Implementations. To efficiently instantiate a circuit that computes AES128, we extensively use lookup tables since an AES128 block cipher is dominated by `Xor`, Galois multiplications and lookups from substitution boxes. All these operations can be performed by using lookup tables. By doing so, an AES cipher can be computed using roughly 2^{10} gates. For proving the SQL query, to perform the range checks and the maximum function, we used the less than operation as presented in [70]. The circuit for the MLP inference is dominated by linear operations except from the activation functions which we implemented using lookup tables. Note that the circuits for proving (c),(d) are data-parallel [5, 63].

[6] As shown in [74], the l chosen in [12] only provides 14-bit security.

Cir. Size	**Space (GB)**		Prove (sec)		Verify (ms)		$\|\pi\|$ (MB)	
$\mathcal{C}=2^{24}$	**0.76**	**46.1**	254	199	44	43	4.3	6
$\mathcal{C}=2^{25}$	**0.96**	**92**	503.8	405.4	61	44	7.3	6.3
$\mathcal{C}=2^{26}$	**1.3**	**121**	998	810.7	94	46	13.3	6.5
$\mathcal{C}=2^{27}$	**2.15**	**x**	1998.4	x	159	x	25.2	x

Thresh. Size	**Space (GB)**	Prove (sec)	Verify (ms)	$\|\pi\|$ (MB)
B=2^{18}	**1**	966.4	158	25.1
B=2^{19}	**1.3**	998	94	13.3
B=2^{20}	**1.8**	1029.2	63	7.4
B=2^{21}	**3**	1055.4	49	4.53

Table 3: Performance of HOBBIT (☐ cells) and its space-inefficient variant (■ cells) when proving an arbitrary circuit on varying circuit size and $B = 2^{19}$ (left) and varying B and constant circuit size $\mathcal{C} = 2^{26}$ (right). "x" indicates memory exhaustion. On the bottom table, the space-inefficient variant of HOBBIT takes 810.7sec to prove, 46sec to verify and has 6.58MB proof size.

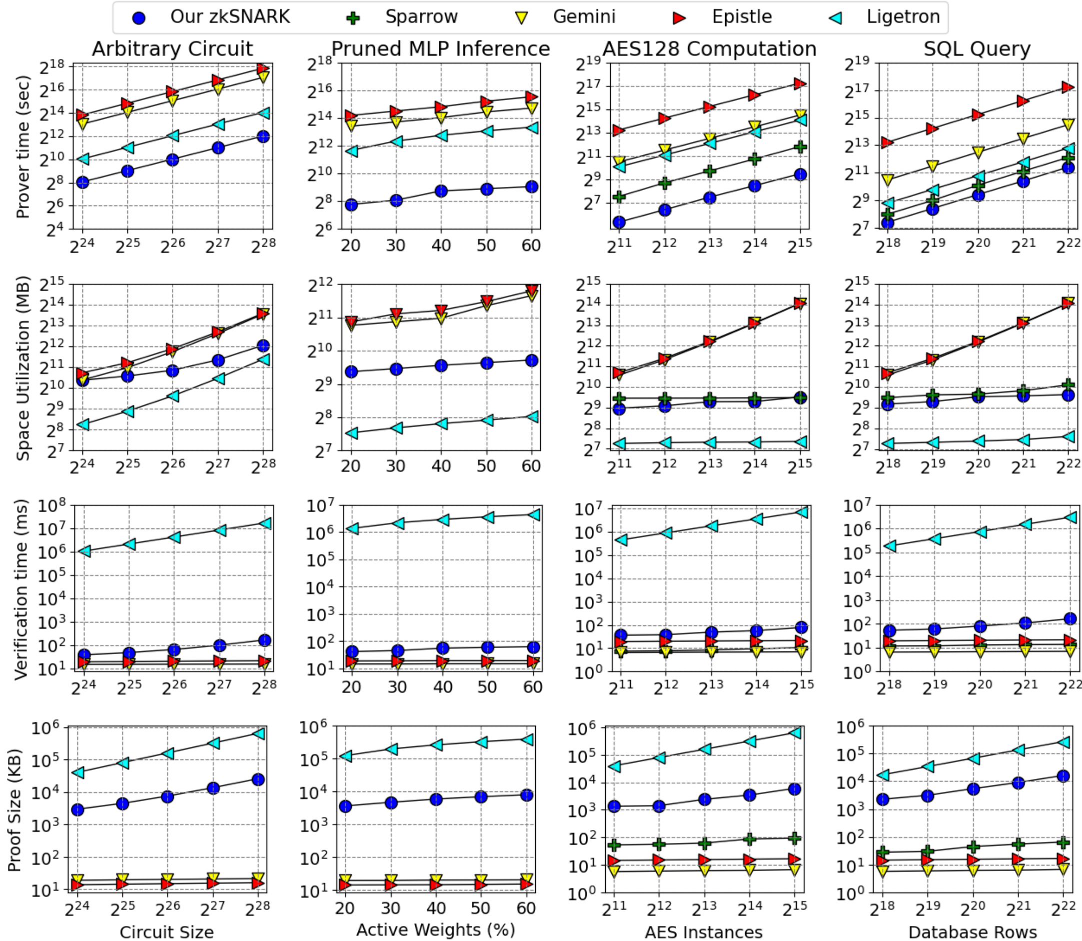

Figure 2: Prover time (far top), space utilization (top), verification time (bottom) and proof size (far bottom) of our zkSNARK, Sparrow, Gemini, Epistle and Ligetron for proving Arbitrary Circuits (far left), Sparse ANN inference (left), multiple AES128 instances (right) and SQL queries (far right). Note that we can not use Sparrow for the first two.

Performance Analysis of HOBBIT. We start by benchmarking HOBBIT when proving an arbitrary arithmetic circuit for: (1) a constant threshold instance $B = 2^{19}$ and varying circuit size ranging from $|\mathcal{C}| = 2^{24}$-2^{27} (Table 3 top) and (2) varying $B = 2^{18}$-2^{21} and constant $|\mathcal{C}| = 2^{26}$ (Table 3 bottom). We also compare HOBBIT with its space-inefficient variant.

From Table 3 (left), observe that the space utilization increases with the increase of the circuit size even though B is constant. This increase solely stems from the evaluation space, which, for the specific circuit we test is $O(|\mathcal{C}|/4)$. In practice, the working buffer space utilized by the prover is constant and occupies 560MB, while the evaluation space increases from 195MB to 1.56GB. A similar trend appears also in Table 3 (right). In particular, because the circuit size is constant, the evaluation space will be the same, namely, 781MB. However, the working buffer space will change from 289MB to 2.26GB as we increase B.

As for prover time, note that it slightly decreases when B becomes smaller. That is because (a) the PCS evaluation time becomes faster with the smaller B and (b) in the sumcheck protocol, the prover has to invoke the standard sumcheck protocol for an instance of size B. Naturally, the smaller B becomes, the less the overhead incurred from the standard sumcheck. Finally, note that the verification time and proof size increase almost linearly with the decrease of B, which agrees with our complexity analysis.

Compared to its space-inefficient variant, HOBBIT uses $\times$60-92 less space. At the same time, our prover is only $\times$1.2 slower! Finally, for most instances, HOBBIT has larger proof sizes and verification times due to its increased complexity.

Comparison with Space-efficient zkSNARKs. We compare our zkSNARK with the state-of-the-art space-efficient zkSNARKs Gemini [19], Epistle [20] and Sparrow [5]. Because Sparrow works only for data-parallel circuits, we can only use it for (c),(d). Moreover, for these applications, we use the variant of Gemini that does not require pre-processing which leads to a significantly faster prover time. Note that since all schemes have a customizable buffer space, we set B to be the same for all of them. This is done for fairness because setting Gemini and Epistle to their buffer space (i.e., $O(\log|\mathcal{C}|)$), would make them significantly slower. With that in mind, we set $B = 2^{19}$ for all applications except for arbitrary circuits where we set $B = 2^{20}$, where they have larger instances. Finally, we also compare HOBBIT with the space-efficient argument Ligetron [21] using its public browser-based version [77] as its source code is not available.

Measuring Space Utilization. We measured the total space needed to generate a proof, namely the working buffer space of the prover and also the space needed to instantiate the streaming oracles. Note that compared to our work, Sparrow [5] and Ligetron [21], Gemini and Epistle follow a slightly different streaming setting from the one described in Section 2.3. In particular, in their code they do not implement their streaming oracles but only "simulate" them by having the prover produce dummy proving data on the fly and only report the working buffer space of the prover. To report the total space used in these protocols we include (1) the space needed to store the public parameters, which is $O(|\mathcal{C}|)$ since both protocols use the KZG scheme and (2) the space for storing the data generated by their streaming oracles.

Prover Time & Space Utilization. Figure 2 reports the prover time (far top) and space utilization (top) of our zkSNARK, Sparrow, Gemini, Epistle and Ligetron for the four applications discussed above. First, observe that our zkSNARK outperforms the existing space-efficient zkSNARKs for arbitrary circuits in proving time and space utilization. Regarding prover time, our zkSNARK is $\times$8.4-32 faster than Gemini and $\times$54.32-56.8 faster than Epistle. At the same time, our zkSNARK uses $\times$1.01-23 and $\times$1.1-23.4 less space than Gemini and Epistle. For instance, to prove an arbitrary arithmetic circuit of size 2^{28}, our zkSNARK prover takes 68min and 4.22GB of memory while Gemini and Epistle require 37.1hrs, 11.9GB and 64.4hrs and 12.2GB respectively. Finally, compared to Ligetron, HOBBIT is $\times$3.9-19.5 faster, but it uses $\times$1.5-4.3 more space. This additional overhead arises from the working buffer space utilized by the provers. Specifically, Ligetron is constrained to a buffer size of $O(\sqrt{|C|})$—smaller than the working buffer space we used in these experiments, resulting in lower space utilization.

For the applications corresponding to data-parallel circuits, our zkSNARK is $\times$1.5-5 faster than Sparrow, which relies on the highly efficient GKR protocol! E.g., to prove the computation of 2^{15} AES128 block ciphers, our zkSNARK takes 702.41sec while Sparrow takes 3543.9sec. Initially, this seems contradicting because zkSNARKs from the GKR protocol are significantly faster than those from HyperPlonk. However, observe that our zkSNARK utilizes lookup arguments which result in significantly smaller circuit sizes compared to those used by Sparrow for the same application. Finally, the space in both schemes is roughly the same, with HOBBIT slightly outperforming Sparrow due to its smaller field size.

Verification Time & Proof Size. Figure 2 reports the verification time (bottom) and proof size (far bottom) of all schemes for the four applications. Compared to the space-efficient zkSNARKs, HOBBIT has slower but still practical verification times ranging between 36-160ms. The one downside of our scheme is the proof size ranging between 1.3-25.3MB. Finally, HOBBIT has at least four orders of magnitude faster verification times than Ligetron. This is expected as Ligetron has a linear-time verifier with respect to the circuit size. At the same time, it also produces $\times$7.6-105.3 smaller proofs!

$\lvert\mathcal{M}\rvert$	**Space (GB)**		Prove (sec)		Verify (ms)		$\lvert\pi\rvert$ (KB)	
0.7M	**0.18**	12	24	274	14	46	693	26.7
1.5M	**0.19**	25	83	568	27	50	1634	26.7
2.6M	**0.31**	50	164	1135	36	48	2418	26.7
4.7M	**0.39**	100	321	2295	53	48	3987	26.7

Table 4: Performance of HOBBIT (□ cells) and EZKL (■ cells) when proving the inference of an MLP model of variable number of parameters $|\mathcal{M}|$. "M" indicates millions.

Testing HOBBIT in a Real-World Application. Finally, we test our scheme in a real-world use case: proving the correct inference of a machine learning model, commonly known as *zkML*. Specifically, we benchmark the performance of HOBBIT when proving the correct inference of a three-layer MLP model for the MNIST classification task [78]. Furthermore, we compare HOBBIT with the state-of-the-art zkML library, EZKL [79], used by both industry [80, 81, 82] and research [83, 84]. Table 4 presents the performance of both schemes when proving the inference of an MLP for parameters ranging between 0.7-4.7M. In general, the space utilization of HOBBIT is significantly smaller than that of EZKL by $\times$68.3-256.4, making it more scalable for larger models. At the same time, our prover time is $\times$7.15-11.9 faster. That is because EZKL uses the Plonk-based Halo2 library [85] which has a quasi-linear prover time and uses expensive elliptic curve operations. Lastly, HOBBIT has comparable or even shorter verification times but produces larger proofs.

Acknowledgments

We thank the anonymous reviewers for their feedback. This work was supported in part by the Hong Kong Research Grants Council under grant GRF-16200721.

Ethics considerations

In this work, we construct a cryptographic scheme, namely, a space-efficient zkSNARK. Our research makes the task of generating a proof accessible to computationally constrained machines, which can be beneficial for securing various systems or protecting the privacy of individuals. We did not use personal data for our experiments, and all our experiments are done on dummy inputs. Furthermore, all cryptographic tools used comply with current laws and regulations.

Open Science

We comply with open science policy by open-sourcing our code and providing the benchmark scripts in https://zenodo.org/records/15620984.

References

[1] J. Kilian, "A note on efficient zero-knowledge proofs and arguments (extended abstract)," in *ACM STOC*, 1992, pp. 723–732.

[2] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE SP*, 2013, pp. 238–252.

[3] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *IEEE SP*, 2014, pp. 459–474.

[4] T. Liu, X. Xie, and Y. Zhang, "zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy," in *CCS*. ACM, 2021, pp. 2968–2985.

[5] C. Pappas and D. Papadopoulos, "Sparrow: Space-efficient zksnark for data-parallel circuits and applications to zero-knowledge decision trees," in *ACM CCS*, 2024, pp. 3110–3124.

[6] K. Abbaszadeh, C. Pappas, J. Katz, and D. Papadopoulos, "Zero-knowledge proofs of training for deep neural networks," in *ACM CCS*, 2024, pp. 4316–4330.

[7] P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, and M. Walfish, "Zero-knowledge middleboxes," in *USENIX Security 2022*, 2022, pp. 4255–4272.

[8] J. Groth, "On the size of pairing-based non-interactive arguments," in *EUROCRYPT*, 2016, pp. 305–326.

[9] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical non-interactive arguments of knowledge," *Cryptology ePrint Archive*, 2019.

[10] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *CRYPTO*, 2019, pp. 733–764.

[11] S. T. V. Setty, "Spartan: Efficient and general-purpose zksnarks without trusted setup," in *CRYPTO*, 2020, pp. 704–737.

[12] T. Xie, Y. Zhang, and D. Song, "Orion: Zero knowledge proof with linear prover time," in *CRYPTO*, 2022, pp. 299–328.

[13] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "Hyperplonk: Plonk with linear-time prover and high-degree custom gates," in *EUROCRYPT*, 2023, pp. 499–530.

[14] J. Zhang, T. Xie, Y. Zhang, and D. Song, "Transparent polynomial delegation and its applications to zero knowledge proof," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 859–876.

[15] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. P. Ward, "Marlin: Preprocessing zksnarks with universal and updatable SRS," in *EUROCRYPT*, 2020, pp. 738–768.

[16] B. Bünz, B. Fisch, and A. Szepieniec, "Transparent snarks from DARK compilers," in *EUROCRYPT*. Springer, 2020, pp. 677–706.

[17] A. R. Block, J. Holmgren, A. Rosen, R. D. Rothblum, and P. Soni, "Public-coin zero-knowledge arguments with (almost) minimal time and space overheads," in *TCC*, 2020, pp. 168–197.

[18] ——, "Time- and space-efficient arguments from groups of unknown order," in *CRYPTO*, 2021, pp. 123–152.

[19] J. Bootle, A. Chiesa, Y. Hu, and M. Orrù, "Gemini: Elastic snarks for diverse environments," in *EUROCRYPT*, 2022, pp. 427–457.

[20] S. Zhang, D. Cai, Y. Li, H. Kan, and L. Zhang, "Epistle: Elastic succinct arguments for plonk constraint system," *IACR Cryptol. ePrint Arch.*, p. 872, 2024.

[21] M. Venkitasubramaniam, "Ligetron: Lightweight scalable end-to-end zero-knowledge proofs. post-quantum zk-snarks on a browser," in *IEEE SP*, 2023, pp. 86–86.

[22] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT 2010*, 2010, pp. 177–194.

[23] C. Papamanthou, E. Shi, and R. Tamassia, "Signatures of correct computation," in *TCC*, 2013, pp. 222–242.

[24] E. W. Allender, "P-uniform circuit complexity," *Journal of the ACM (JACM)*, vol. 36, no. 4, pp. 912–928, 1989.

[25] P. Valiant, "Incrementally verifiable computation or proofs of knowledge imply time/space efficiency," in *TCC*, 2008, pp. 1–18.

[26] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "Recursive composition and bootstrapping for SNARKS and proof-carrying data," in *STOC*. ACM, 2013, pp. 111–120.

[27] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable zero knowledge via cycles of elliptic curves," in *CRYPTO*, 2014, pp. 276–294.

[28] A. Kothapalli, S. T. V. Setty, and I. Tzialla, "Nova: Recursive zero-knowledge arguments from folding schemes," in *CRYPTO*, 2022, pp. 359–388.

[29] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner, "Proof-carrying data without succinct arguments," in *CRYPTO*, 2021, pp. 681–710.

[30] S. Bowe, J. Grigg, and D. Hopwood, "Halo: Recursive proof composition without a trusted setup," *IACR Cryptol. ePrint Arch.*, p. 1021, 2019.

[31] W. D. Nguyen, T. Datta, B. Chen, N. Tyagi, and D. Boneh, "Mangrove: A scalable framework for folding-based snarks," in *CRYPTO*, 2024, pp. 308–344.

[32] B. Bünz and J. Chen, "Proofs for deep thought: Accumulation for large memories and deterministic computations," in *ASIACRYPT*, 2024, pp. 269–301.

[33] M. Chen, A. Chiesa, T. Gur, J. O'Connor, and N. Spooner, "Proof-carrying data from arithmetized random oracles," in *EUROCRYPT*, 2023, pp. 379–404.

[34] M. Chen, A. Chiesa, and N. Spooner, "On succinct non-interactive arguments in relativized worlds," in *EUROCRYPT*, O. Dunkelman and S. Dziembowski, Eds. Springer, 2022, pp. 336–366.

[35] H. Lee and J. H. Seo, "On the security of nova recursive proof system," *IACR Cryptol. ePrint Arch.*, p. 232, 2024.

[36] J. Bootle, A. Chiesa, and J. Groth, "Linear-time arguments with sublinear verification from tensor codes," in *TCC*, 2020, pp. 19–46.

[37] "https://github.com/ChristodoulosPappas/HOBBIT-Space-Efficient-zkSNARK-with-Optimal-Prover-Time."

[38] M. Brehm, B. Chen, B. Fisch, N. Resch, R. D. Rothblum, and H. Zeilberger, "Blaze: Fast snarks from interleaved RAA codes," in *EUROCRYPT*, 2025, pp. 123–152.

[39] H. Zeilberger, B. Chen, and B. Fisch, "Basefold: Efficient field-agnostic polynomial commitment schemes from foldable codes," in *CRYPTO*. Springer, 2024, pp. 138–169.

[40] R. Canetti, Y. Chen, J. Holmgren, A. Lombardi, G. N. Rothblum, and R. D. Rothblum, "Fiat-shamir from simpler assumptions," *IACR Cryptol. ePrint Arch.*, p. 1004, 2018.

[41] R. Canetti, Y. Chen, J. Holmgren, A. Lombardi, G. N. Rothblum, R. D. Rothblum, and D. Wichs, "Fiat-shamir: from practice to theory," in *STOC*, 2019, pp. 1082–1090.

[42] E. Ben-Sasson, A. Chiesa, and N. Spooner, "Interactive oracle proofs," in *TCC*, 2016, pp. 31–60.

[43] J. Holmgren, "On round-by-round soundness and state restoration attacks," *IACR Cryptol. ePrint Arch.*, p. 1261, 2019.

[44] A. J. Blumberg, J. Thaler, V. Vu, and M. Walfish, "Verifiable computation using multiple provers," *Cryptology ePrint Archive*, 2014.

[45] A. Chiesa, E. Fedele, G. Fenzi, and A. Zitek-Estrada, "A time-space tradeoff for the sumcheck prover," *Cryptology ePrint Archive*, 2024.

[46] A. Kothapalli and S. T. V. Setty, "Neutronnova: Folding everything that reduces to zero-check," *IACR Cryptol. ePrint Arch.*, p. 1606, 2024.

[47] V. Nair, A. Sharma, and B. Thankey, "Brakingbase-a linear prover, poly-logarithmic verifier, field agnostic polynomial commitment scheme," *Cryptology ePrint Archive*, 2024.

[48] A. Golovnev, J. Lee, S. T. V. Setty, J. Thaler, and R. S. Wahby, "Brakedown: Linear-time and field-agnostic snarks for R1CS," in *CRYPTO*, 2023, pp. 193–226.

[49] D. A. Spielman, "Linear-time encodable and decodable error-correcting codes," *IEEE Trans. Inf. Theory*, vol. 42, no. 6, pp. 1723–1731, 1996. [Online]. Available: https://doi.org/10.1109/18.556668

[50] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[51] N. Ron-Zewi and R. D. Rothblum, "Proving as fast as computing: succinct arguments with constant prover overhead," in *STOC*, 2022, pp. 1353–1363.

[52] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *IEEE SP*, 2018, pp. 315–334.

[53] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, “Ligero: Lightweight sublinear arguments without a trusted setup,” in *ACM CCS*, 2017, pp. 2087–2104.

[54] L. Bangalore, R. Bhadauria, C. Hazay, and M. Venkitasubramaniam, “On black-box constructions of time and space efficient sublinear arguments from symmetric-key primitives,” in *TCC*, 2022, pp. 417–446.

[55] A. Baweja, P. Mishra, T. Mopuri, K. Newatia, and S. Wang, “Scribe: Low-memory snarks via read-write streaming,” *IACR Cryptol. ePrint Arch.*, p. 1970, 2024.

[56] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan, “Algebraic methods for interactive proof systems,” *J. ACM*, pp. 859–868, 1992.

[57] N. Bitansky and A. Chiesa, “Succinct arguments from multi-prover interactive proofs and their efficiency benefits,” in *CRYPTO*, 2012, pp. 255–272.

[58] J. Zhang, T. Liu, W. Wang, Y. Zhang, D. Song, X. Xie, and Y. Zhang, “Doubly efficient interactive proofs for general arithmetic circuits with linear prover time,” in *CCS*. ACM, 2021, pp. 159–177.

[59] D. Balbás, D. Fiore, M. I. G. Vasco, D. Robissout, and C. Soriente, “Modular sumcheck proofs with applications to machine learning and image processing,” in *CCS*. ACM, 2023, pp. 1437–1451.

[60] G. Arnon, A. Chiesa, G. Fenzi, and E. Yogev, “WHIR: reed-solomon proximity testing with super-fast verification,” in *EUROCRYPT*, 2025, pp. 214–243.

[61] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, “Delegating computation: interactive proofs for muggles,” in *ACM STOC*, C. Dwork, Ed., 2008, pp. 113–122.

[62] D. Khovratovich, R. D. Rothblum, and L. Soukhanov, “How to prove false statements: Practical attacks on fiat-shamir,” *IACR Cryptol. ePrint Arch.*, p. 118, 2025.

[63] J. Thaler, “Time-optimal interactive proofs for circuit evaluation,” in *CRYPTO*, 2013, pp. 71–89.

[64] A. R. Block, A. Garreta, P. R. Tiwari, and M. Zajac, “On soundness notions for interactive oracle proofs,” *J. Cryptol.*, vol. 38, no. 1, p. 4, 2025. [Online]. Available: https://doi.org/10.1007/s00145-024-09520-7

[65] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor, “Checking the correctness of memories,” in *FOCS*. IEEE Computer Society, 1991, pp. 90–99.

[66] S. T. V. Setty, J. Thaler, and R. S. Wahby, “Uwhirnlocking the lookup singularity with lasso,” in *EUROCRYPT*, 2024, pp. 180–209.

[67] S. T. V. Setty and J. Lee, “Quarks: Quadruple-efficient transparent zksnarks,” *IACR Cryptol. ePrint Arch.*, 2020.

[68] S. T. V. Setty, J. Thaler, and R. S. Wahby, “Customizable constraint systems for succinct arguments,” *IACR Cryptol. ePrint Arch.*, p. 552, 2023.

[69] L. Pearson, J. B. Fitzgerald, H. Masip, M. Bellés-Muñoz, and J. L. Muñoz-Tapia, “Plonkup: Reconciling plonk with plookup,” *IACR Cryptol. ePrint Arch.*, p. 86, 2022.

[70] A. Arun, S. T. V. Setty, and J. Thaler, “Jolt: Snarks for virtual machines via lookups,” in *EUROCRYPT*, 2024, pp. 3–33.

[71] Https://github.com/BLAKE3-team/BLAKE3.

[72] “https://github.com/0xpolygonzero/plonky2.”

[73] U. Haböck, “Brakedown’s expander code,” *Cryptology ePrint Archive*, 2023.

[74] T. den Hollander and D. Slamanig, “A crack in the firmament: Restoring soundness of the orion proof system and more,” *IACR Cryptol. ePrint Arch.*, p. 1164, 2024.

[75] C. Ding and Y. Huang, “Dubhe: Succinct zero-knowledge proofs for standard AES and related applications,” in *USENIX Sec*, 2023, pp. 4373–4390.

[76] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, “vsql: Verifying arbitrary SQL queries over dynamic outsourced databases,” in *IEEE SP*, 2017, pp. 863–880.

[77] “https://platforms.ligetron.com/marketplace.”

[78] L. Deng, “The mnist database of handwritten digit images for machine learning research [best of the web],” *IEEE signal processing magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[79] “https://ezkl.xyz/.”

[80] “https://blog.ezkl.xyz/post/state_of_ezkl/.”

[81] “https://ritual.net/.”

[82] “https://www.opengradient.ai/.”

[83] C. Yadav, E. M. Laufer, D. Boneh, and K. Chaudhuri, “Expproof: Operationalizing explanations for confidential models with zkps,” *arXiv preprint arXiv:2502.03773*, 2025.

[84] P. Germani, M. A. Manzari, R. Magni, P. Dibitonto, F. Previtali, and E. D’Agostini, “Building trustworthy ai systems: Ai inference verification with blockchain and zero-knowledge proofs,” in *2024 6th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 2024, pp. 1–3.

[85] “https://github.com/zcash/halo2.”

[86] F. Hirner, F. Krieger, C. Piber, and S. S. Roy, “Orion’s ascent: Accelerating hash-based zero knowledge proof on hardware platforms,” *IACR Cryptol. ePrint Arch.*, p. 1918, 2024.

[87] “https://github.com/sunblaze-ucb/Orion/blob/linearPC/src/linear_gkr/verifier.cpp.”

A Issues of HyperPlonk and R1CS when used directly in the Streaming Setting

Assume a type of arbitrary looking circuit, the representation of which can be generated on the fly using a small space. Such circuits, which fall under the category of log-space uniform circuits, are commonly used in practice [86]. Here, we will illustrate the challenges that arise when trying to prove these circuits with Epistle [20] and Gemini [19] while relying solely on streaming access to $S(\mathbf{tr})$. To demonstrate this, we will consider a scenario in which there is a gate labeled v that is used twice; once at the beginning and once at the end of the computation.

Starting from the Plonk constraint system, recall that, to prove wiring consistency, the prover has to show that $\prod_{\mathbf{i}\in\{0,1\}^{\log N}}(f(\mathbf{i})+i) = \prod_{\mathbf{i}\in\{0,1\}^{\log N}}(f(\mathbf{i})+\sigma(\mathbf{i}))$. The main issue arises when we try to instantiate streaming access to the coefficients of σ. In particular, assume that j is the index of the first output wire of v. When proving the latter equation, the prover must also compute $\sigma(\mathbf{j})$. To achieve this, it has to generate the representation circuit on the fly until it finds the gate where the output of v is used again. Since $\sigma(\mathbf{j})$ appears at the end of the circuit, this would require $O(|\mathcal{C}|)$ time. In the worst case, the circuit has more such gates, and consequently a single pass over $S(\sigma)$, would take $\Omega(N)$ time.

A similar issue also arises when using the R1CS system. In more detail, in Gemini, the prover must have streaming access to the columns of the R1CS constraint matrixes A,B,C (e.g., to prove the second sumcheck of [11] in the streaming setting [19]). In particular, for any matrix $G \in \{A,B,C\}$ and for each column, $S(G)$ outputs the row indexes with non-zero values. Since each column represents a gate, identifying all such row indices necessitates the streaming oracle to parse the entire circuit by generating it on the fly. Following the same reasoning as with Plonk to show that a pass over $S(G)$ requires $\Omega(N)$ time. Lastly, we note that in both examples, one could avoid the increase in prover time with the cost of increasing its space to $O(|\mathcal{C}|)$.

B Open-Source Implementation of Orion

In the implementation of Orion, the authors use a different PCS following the “Brakedown + proof composition” approach, but set the number of rows to 128. Compared to BrakingBase and our PCS, they use the GKR protocol directly to prove the correct computation of the encoding process. Although this can lead to a very fast prover time it results $O(N)$ verification time. That is because the circuit that represents the encoding algorithm does not have a repetitive structure. For these types of circuits the GKR incurs a linear verification time [63]. Note that in order to make it succinct, it would require to commit to the expander graph leading to a similar protocol with BrakingBase. Additionally, although the authors measure the verification time of the GKR protocol in their implementation, they do not include this in the experimental evaluation of their paper (see their comment in line 1237 of [87]). Finally, their scheme is roughly $\times 2$ slower than the non-succinct variant of Orion but it has $\times 1.6$ smaller proofs and $\times 10$ faster verifier.