

SPARROW: Space-Efficient zkSNARK for Data-Parallel Circuits and Applications to Zero-Knowledge Decision Trees

Christodoulos Pappas*

Dimitrios Papadopoulos*

Abstract

Space-efficient SNARKs aim to reduce the prover’s space overhead which is one the main obstacles for deploying SNARKs in practice, as it can be prohibitively large (e.g., orders of magnitude larger than natively performing the computation). In this work, we propose SPARROW, a novel space-efficient zero-knowledge SNARK for data-parallel arithmetic circuits with two attractive features: (i) it is the first space-efficient scheme where, for a given field, the prover overhead increases with a multiplicative *sublogarithmic* factor as the circuit size increases, and (ii) compared to prior space-efficient SNARKs that work for arbitrary arithmetic circuits, it achieves prover space *asymptotically smaller* than the circuit size itself. Our key building block is a novel space-efficient sumcheck argument with improved prover time which may be of independent interest. Our experimental results for three use cases (*arbitrary data parallel circuits, multiplication trees, batch SHA256 hashing*) indicate SPARROW outperforms the prior state-of-the-art space-efficient SNARK for arithmetic circuits GEMINI (Bootle et al., EUROCRYPT’22) by 3.2-28.7× in total prover space and 3.1-11.3× in prover time. We then use SPARROW to build *zero-knowledge proofs of tree training and prediction*, relying on its space efficiency to scale to large datasets and forests of multiple trees. Compared to a (non-space-efficient) optimal-time SNARK based on the GKR protocol, we observe prover space reduction of 16-240× for tree training while maintaining essentially the same prover and verifier times and proof size. Even more interestingly, *our prover requires comparable space to natively performing the underlying computation*. E.g., for a 400MB dataset, our prover only needs 1.4× more space than the native computation.

1 Introduction

Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge [1, 2, 3] (zkSNARKs) enable a *prover* to convince a *verifier* about the validity of an NP statement by only sending a small & easy-to-verify proof, and without revealing additional information. Over the past years, zkSNARKs evolved from a theoretical concept to integral parts of securing several applications such as anonymous cryptocurrencies [4], scaling and bridging blockchains [5, 6], and ensuring the integrity of machine learning tasks [7, 8]. A long series of works have significantly improved the (considerable) prover’s runtime both asymptotically and concretely [3, 9, 10, 11, 12, 13, 14], however, modern zkSNARKs share a common bottleneck: *excessive prover space utilization*. Simply put, the space (typically memory) needed to prove computations scales to orders of magnitude more than natively running the computation, often making it impossible to run for realistically large problem instances. To put numbers in perspective, proving the prediction of a VGG16 model using the highly efficient zkSNARK of [8] takes 24GB of memory, while the plaintext computation needs < 200MB—two orders of magnitude less. Likewise, proving SHA256 for a pre-image of 16KB with Plonk [9] and Groth16 [3] takes space of 128GB and 40GB, respectively [15].

*Hong Kong University of Science and Technology, cpappas@connect.ust.hk, dipapado@cse.ust.hk

Scheme	Model	Prove	Verify	$ \pi $	Buffer	Streams	pp	Total Space
Block et al. [19]	RAM	$T \cdot \text{polylog } T$	$T \log T$	$\log T$	$S \cdot \text{polylog } T$	$S \cdot \text{polylog } T$	1	$S \cdot \text{polylog } T$
Block et al. [21]	RAM	$T \cdot \text{polylog } T$	$ x \cdot \text{polylog } T$	$\log T$	$S \cdot \text{polylog } T$	$S \cdot \text{polylog } T$	1	$S \cdot \text{polylog } T$
Ligetron [22]	Arithm. Circ.	$ \mathcal{C} \log \mathcal{C} $	$ \mathcal{C} $	$\sqrt{ \mathcal{C} }$	$\sqrt{ \mathcal{C} }$	S_{eval}	1	$\sqrt{ \mathcal{C} } + S_{eval}$
GEMINI [26]	Arithm. Circ.	$ \mathcal{C} \log^2 \mathcal{C} $	$ x + \log \mathcal{C} $	$\log \mathcal{C} $	$\log \mathcal{C} $	S_{eval}	$ \mathcal{C} $	$ \mathcal{C} $
Sparrow	LDP Arithm. Circ.	$ \mathcal{C} \log \log \mathcal{C} $	$ x + \log \mathcal{C} $	$\log \mathcal{C} $	$\sqrt{ \mathcal{C} }$	$ \mathcal{C}' + \text{inp}(\mathcal{C}) $	$\sqrt{ \mathcal{C} }$	$\sqrt{ \mathcal{C} } + \text{inp}(\mathcal{C}) $

Table 1: Asymptotic comparison of works on space-efficient arguments. For a RAM program, T is the number of execution steps, and S is the memory size. For an arithmetic circuit \mathcal{C} , $\text{inp}(\mathcal{C})$ is its input, and S_{eval} denotes the space needed for its evaluation ($\mathcal{O}(|\mathcal{C}|)$ for arbitrary circuits). x is the statement being proven by the argument. A layered data-parallel arithmetic (LDP) circuit \mathcal{C} consists of parallel copies of a circuit \mathcal{C}' with $|\mathcal{C}'| \ll |\mathcal{C}|$. The provers of Gemini and SPARROW also require MSM of size $\mathcal{O}(|\mathcal{C}'|)$ for their PC schemes. The schemes of [19, 22] are not succinct.

Space-efficient arguments try to address this issue by ensuring the prover’s time and space asymptotically remain as close as possible to those of the actual computation. Early works in this area study the problem in more relaxed settings, e.g., designated verifier arguments [16, 17] or schemes with multiple non-colluding provers [18]. The first publicly verifiable space-efficient argument was proposed by Block et al. [19], working in the RAM model of computation. As with most modern arguments of knowledge, the authors of [19] first develop a *polynomial interactive oracle proof* (PIOP) [20] (inspired by [18]) that proves the correct execution of each RAM instruction and then compile it into an argument of knowledge using a *polynomial commitment (PC)* scheme. Contrary to prior work, however, they develop both schemes *in the streaming setting* in which the prover uses only a small working buffer space and has streaming (read-only) access to the necessary data required to generate a proof. In this setting, that was adopted in subsequent works [19, 21, 22, 23], including ours, space complexity is measured by the space required for (1) the working buffer space of the prover (2) instantiating access to the streaming oracles and (3) storing public parameters (e.g., for the PC scheme). Assuming a RAM program running in T steps and using space S , [19] achieves space complexities of $\mathcal{O}(S \cdot \text{polylog } T)$, $\mathcal{O}(S \cdot \text{polylog } T)$ and $\mathcal{O}(1)$ respectively for (1)-(3), leading to a total space complexity of $\mathcal{O}(S \cdot \text{polylog } T)$. However, that scheme is not succinct as their PC scheme relies on Bulletproofs [24] and has quasi-linear verification time. In subsequent work, Block et al. [21] improved this, introducing the first space-efficient zkSNARK for RAM computations, based on a novel space-efficient PC scheme on hidden-order groups.

Subsequent works moved from the RAM model to *arithmetic circuits*, thus avoiding the necessary overhead of “translating” RAM computations to circuits. Very recently, Wang et. al. [22] proposed Ligetron, an MPCitH-based [25] space-efficient argument of knowledge which relies on the ideas of [23]. For an arithmetic circuit \mathcal{C} with optimal-evaluation space S_{eval} , Ligetron achieves proving complexity of $\mathcal{O}(|\mathcal{C}| \log |\mathcal{C}|)$ and space complexity of $\mathcal{O}(S_{eval} + \sqrt{|\mathcal{C}|})$. The latter derives from the fact that Ligetron needs $\mathcal{O}(\sqrt{|\mathcal{C}|})$ working buffer space, $\mathcal{O}(S_{eval})$ space to instantiate access to the streaming oracles and $\mathcal{O}(1)$ space to store cryptographic parameters. Unfortunately, it is also *not succinct* since the proof size and verification time are $\mathcal{O}(\sqrt{|\mathcal{C}|})$ and $\mathcal{O}(|\mathcal{C}|)$, respectively.

Bootle et al. [26] focus on a slightly different setting with *external streaming oracles* via which the prover accesses the necessary data and public parameters. Considering, for instance, an external party with enough space to store or generate these streams, they ignore the space requirements

of (2),(3) and focus on *minimizing* the working buffer space (1). Indeed, they achieve $\mathcal{O}(\log |\mathcal{C}|)$ working buffer space as follows. First, they introduce a space-efficient PIOP for the satisfiability of the R1CS constraint system to encode \mathcal{C} . Then, they compile this into a space-efficient zkSNARK called GEMINI, by designing space-efficient variants of the commit and evaluation algorithms of the KZG PC scheme [27] operating with small buffer space. However, when measuring the total space complexity including space to instantiate streams and store public parameters, as in [19, 21, 22, 23] and our work, GEMINI has total space complexity $\mathcal{O}(|\mathcal{C}|)$. That is because it uses the KZG scheme, which needs public parameters of size $\mathcal{O}(|\mathcal{C}|)$. Furthermore, the space needed to instantiate the streams is $\mathcal{O}(S_{eval})$ space; for arbitrary arithmetic circuits, this can also be $\mathcal{O}(|\mathcal{C}|)$. Finally, GEMINI is *elastic*, i.e., it allows the user to increase the working buffer space (beyond $\mathcal{O}(\log |\mathcal{C}|)$) to improve the prover time, as a trade-off between space and time.

Table 1 shows the asymptotic performance of these works and offers two observations. (1) All constructions achieve prover complexity that scales quasi-linearly with respect to the circuit size (for succinct schemes, that is no less than $\mathcal{O}(|\mathcal{C}| \log^2 |\mathcal{C}|)$). This is in contrast to non-space efficient SNARKs, e.g., [10, 11, 12] that achieve prover complexity *scaling linearly to* $|\mathcal{C}|$. (2) An inherent limitation when working with arbitrary circuits is that optimal evaluation space—and hence proving space—takes $\mathcal{O}(|\mathcal{C}|)$. These lead to the following question: “*Can we have a space-efficient zkSNARK that, for a “rich” class of arithmetic circuits, achieves (almost) optimal prover time, and prover space asymptotically smaller than $|\mathcal{C}|$?*”

Sparrow: Space-Efficient zkSNARK for Data-Parallel Circuits. In this work, we answer the above question by presenting SPARROW, a space-efficient zkSNARK that works for *layered data-parallel arithmetic* circuits (LDP). This class of circuits can model a wide variety of real-world computational tasks and has been widely used in the literature to build (non-space efficient) zkSNARKs, with applications ranging from SQL database queries [28], blockchain L2 rollups and blockchain bridges [5, 6], training and prediction of machine learning models [7, 29, 8] and verifiable symmetric key encryption [30]. SPARROW has a prover that requires only $\mathcal{O}(|\mathcal{C}| \log \log |\mathcal{C}|)$ field operations, (and multi-scalar multiplications (MSM) of size $\mathcal{O}(|\mathcal{C}|)$, for the PC) and verification complexity and proof size of $\mathcal{O}(\log |\mathcal{C}|)$. The prover’s space is only $\mathcal{O}(\sqrt{|\mathcal{C}|} + |\text{inp}(\mathcal{C})|)$ where $\text{inp}(\cdot)$ is the input of the circuit.

Our main building block is a novel space-efficient sumcheck argument which, given streaming access to the coefficients of multi-linear polynomials $f, g : \mathbb{F}^n \rightarrow \mathbb{F}$, proves the instance $K = \sum_{x \in \{0,1\}^n} f(x)g(x)$. Prior works [21, 26] use a similar protocol [18] but it has $\mathcal{O}(N \log N)$ proving time, where $N = 2^n$, since for every round (for $i = 1, \dots, \log N$), it must scan the entire polynomials. We adopt a different approach, “reducing” the number of variables of f, g by replacing them with equivalent multi-variate polynomials of higher degree. In that sense, our space-efficient sumcheck can be considered as a hybrid between the multi-linear [31] and the univariate sumcheck protocol [32]. Our protocol requires $\mathcal{O}(N \log \log N)$ field operations and $\mathcal{O}(\sqrt{N})$ buffer space. Concurrent work by Chiesa et al. [33] proposed a sumcheck with the same space as ours and linear prover time. However, it only supports instances of the form $K = \sum_{x \in \{0,1\}^n} f(x)$, but not the products-of-polynomials that are needed for PIOPs (like the one in SPARROW).

Armed with our novel sumcheck, we build a space-efficient PIOP and compile it into the SPARROW zkSNARK via a space-efficient PC scheme (similar to [21, 26]). For the first, we propose a space-efficient variant of the GKR [34] protocol. Besides replacing the standard sumcheck protocol with our space-efficient one, we also (i) efficiently instantiate streaming access to the evaluation circuit layers, and (ii) “flatten” the circuit [10, 18], if it is deep, to avoid increasing overheads with its depth, while keeping its data-parallel structure. Finally, we adapt the optimal-time space-efficient variant of the “Dory-like” [35] PC, Kopsis [36], to work with $\mathcal{O}(\sqrt{N})$ buffer and public parameters size (in contrast to $\mathcal{O}(N)$ of [26]).

Zero-knowledge Proofs of Forest Training. With the widespread adoption of machine learning (ML) algorithms in domains like business decision-making and healthcare, there is an increasing demand to boost the public’s confidence in them [37, 38]. A crucial requirement to achieve this is *integrity*, e.g., the ability to prove that model training and predictions have performed honestly. Recent works propose using zero-knowledge proofs to ensure such integrity guarantees e.g., for ML prediction of neural networks and decision trees [8, 39, 40, 41]. On the other hand, zero-knowledge proofs for ML training have been studied much less and only for neural networks [7] and linear regression [42]. Especially training is an inherently space-demanding task, as it works on potentially massive datasets (as opposed to prediction that works on the model).

In this work, we specifically focus on decision tree-based models [43, 44], which remain one of the most widely used ML tools in practice [45, 46], and introduce the first scheme for *zero-knowledge proofs of forest training and predictions* (*zkFTP*). It allows a prover to commit to a dataset D and a forest \mathcal{F} and provide zkSNARK proofs that (committed) \mathcal{F} was correctly trained over (committed) D and that y is the correct prediction for a test point \mathbf{x} . To make our *zkFTP* scalable to large instance sizes, we use our space-efficient zkSNARK SPARROW as its back-end for proofs of training.

To further improve the performance of the latter, instead of encoding the training into an arithmetic circuit, we propose a *certification algorithm* (Section 4.1) that validates the correctness of a model for a dataset asymptotically faster than training it. We apply further optimizations when encoding our certification algorithm into a data-parallel circuit, exploiting the homomorphic properties of node histograms and offline memory-checking techniques [11]. For predictions (a naturally more “lightweight” task since they operate on the trees and not the massive dataset), we rely on the previous efficient schemes of [39, 40], making our *zkFTP* compatible with them to achieve “end-to-end” security for the ML pipeline.

Experimental Evaluation. We implemented SPARROW and *zkFTP* in C++ and experimentally evaluated their performance. As with GEMINI, our implementation is also elastic. The user can configure the working buffer space by selecting a threshold instance size. When instances become smaller than that threshold, the prover automatically changes from space-efficient to time-efficient, running the standard GKR-based zkSNARK of [10]. We evaluated three different use cases: (a) *arbitrary data-parallel circuits*, (b) *multiplication trees*, and (c) *batch SHA256 hashing*.

Our experimental results (Section 5) show that SPARROW achieves significantly lower space usage and faster prover than prior succinct space-efficient SNARKs for arithmetic circuits. In particular, SPARROW outperforms GEMINI in prover time by 3.4-9.5 \times . When comparing the schemes, we assign the same threshold instance size 2^{20} (the largest instance we ran is 2^{36}), to ensure fair comparison. Although we could set GEMINI’s threshold much lower (close to $\log |\mathcal{C}|$), thereby reducing its buffer space, this would make its prover significantly slower. As for *total* space usage, SPARROW achieves 14.5-28.7 \times space reduction. Putting numbers into perspective, to prove 2048 SHA hashes SPARROW takes only 700MB and 13min vs. 10.5GB and 46min for GEMINI; for 2^{30} -sized LDP circuits, the corresponding numbers are 2.7GB and 78min for SPARROW and 80GB and 744min for GEMINI (see section 5.1 for more details).

The only drawback of SPARROW is its slightly larger proofs and verification times which, however, remain concretely practical: our maximum proof size and verification time in all use cases were $< 90\text{KB}$ and $< 15\text{ms}$. We also compared SPARROW with a non-space-efficient variant based on GKR and the PC of [36] to demonstrate its scalability. Overall, SPARROW achieves 27.5-119 \times space reduction. Interestingly, while scaling to much larger instance sizes (for large instances the non-space-efficient SNARK ran out of memory on our machine with 131GB RAM), SPARROW achieves at most a 2.7 \times slowdown in prover time vs. the non-space-efficient zkSNARK.

We used datasets generated via `make_classification` from `scikit-learn` library to benchmark our *zkFTP*, with variable number of points n between 2^{14} - 2^{22} , and number of features d between

8-64. We quantized our datasets (bin size 128) and considered forest sizes K between 1-128 trees. In this setting, again, our results are very promising. Compared to the non-space-efficient zkSNARK described above, our SPARROW-based zkFTP achieves 16-240 \times space reduction. Moreover, even for medium-sized datasets, we were not able to run the non-space-efficient zkSNARK for training forests. In contrast SPARROW’s space scales strictly linearly to the dataset. For instance, for the largest instance ($n = 2^{20}$, $d = 16$, $K = 1$) for which we could run the non-space-efficient zkSNARK, SPARROW takes 0.42GB (vs. 90GB for the non-space-efficient one), and maintains almost the same prover and verification times and proof size. Finally, for large enough instances, SPARROW’s prover requires space *concretely comparable* to that of directly running forest certification! E.g., to train a single tree using a dataset with $n = 2^{22}$, $d = 16$ (totaling to 400MB), running our certification takes 676MB, whereas SPARROW takes 950MB to produce a proof—just a 1.4 \times increase.

Other related works. A different line of works builds zkSNARKs [47, 48, 7, 49, 50, 51, 52, 53] via *recursive proof composition* with applications to incremental/streaming computation and proof-carrying data. While not their main goal, a side-effect of breaking down a computation into steps and recursively proving them is that prover’s memory usage can, in principle, be made as small as one computation step. However, this approach has some crucial limitations. Performance-wise, recursion leads to significant overheads [7, 54] (e.g., to embed the verifier logic in the circuit), or to non-succinct schemes [49, 51, 52]. Furthermore, works that do not have straight-line extraction [55] cannot be shown secure for more than a constant number of recursions. Finally, most efficient schemes rely on a hash function that must be modeled as a random oracle. During recursion, this “random oracle” must be encoded to the circuit and used in a non-black-box way by the recursive verifier, which makes it impossible to formally argue about security.

Prior works on zero-knowledge for ML can be split into two categories. Proofs of training, for neural networks [7] and linear regression [42], and proofs of prediction, e.g., for neural networks [56, 57, 58, 8], for linear and logistic regression [59], for SVM [60], and for decision trees [39, 40]. For prediction, our *zkFTP* uses the approach of [39] and the matrix-lookup argument of [40]. To the best of our knowledge, no prior work tries to prove the training of a tree or random forest. Finally, non-zero-knowledge approaches for proving model training based on spot random checks [59, 61] or partial re-execution [62] do not achieve the strong security guarantees we aim for (in fact the latter has been shown insecure [63]).

2 Preliminaries

We denote $[n] = \{1, \dots, n\}$. Let \mathbb{F} be a field of prime order p . We use $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{F}^n$ to represent a vector and \mathbf{x}_i or $\mathbf{x}[i]$ its i -th element. We can encode any vector $\mathbf{x} : \{0, 1\}^{\log n} \rightarrow \mathbb{F}$ as a multi-linear polynomial $f_{\mathbf{x}} : \mathbb{F}^{\log n} \rightarrow \mathbb{F}$, s.t. $f_{\mathbf{x}}(\mathbf{z}) = \sum_{i \in \{0, 1\}^{\log n}} \beta(\mathbf{z}, \mathbf{i}) \mathbf{x}_i$, where $\beta(\mathbf{z}, \mathbf{i}) = \prod_{k \in [\log n]} (\mathbf{i}_k \mathbf{z}_k + (1 - \mathbf{i}_k)(1 - \mathbf{z}_k))$. We say that $f_{\mathbf{x}}(\mathbf{z})$ is the multi-linear extension of \mathbf{x} . We denote with \mathbb{H} , a multiplicative sub-group of \mathbb{F} , and by $L_i(x)$, where $\omega^i \in \mathbb{H}$, the Lagrange polynomial such that $L_i(\omega^i) = 1$ and $L_i(\omega^j) = 0, \forall j \in [\mathbb{H}] \setminus i$. Finally, \otimes denotes the tensor product of two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{F}^n$ defined as $\mathbf{a} \otimes \mathbf{b} = (\mathbf{a}_1 \mathbf{b}_1, \dots, \mathbf{a}_1 \mathbf{b}_n, \dots, \mathbf{a}_n \mathbf{b}_1, \dots, \mathbf{a}_n \mathbf{b}_n) \in \mathbb{F}^{n^2}$.

2.1 Interactive Proofs

An *interactive proof* system [64, 34, 65, 31] is run between a *prover* and a *verifier*, on shared input x and language \mathcal{L} , to establish $x \in \mathcal{L}$. We say it is *sound* if the verifier rejects any claim where $x \notin \mathcal{L}$ with high probability and *complete* if the verifier accepts when $x \in \mathcal{L}$. More formally:

Definition 2.1 (Interactive Proofs) *An interactive proof system for a language \mathcal{L} with soundness ϵ , is a protocol between a computationally unbounded prover \mathcal{P} and a probabilistic polynomial time*

verifier \mathcal{V} in which both parties share a common input x . Assuming that $t = \langle \mathcal{P}, \mathcal{V} \rangle(x)$ is the communication transcript produced by the interaction between \mathcal{P} and \mathcal{V} , an interactive proof system satisfies the following:

- **Completeness.** For every $x \in \mathcal{L}$, $\Pr(V(x, t) = \text{accept}) = 1$.
- **Soundness.** For every $x \notin \mathcal{L}$, $\Pr(V(x, t) = \text{accept}) \leq \epsilon$

Throughout the paper, we will rely on the following interactive proofs:

Sumcheck Protocol. The sumcheck protocol of Lund et al. [31] is an interactive proof where, given an n -variable polynomial of degree d defined over a finite field \mathbb{F} , $f : \mathbb{F}^n \rightarrow \mathbb{F}$, the prover wishes to convince the verifier that $K = \sum_{x_1, x_2, \dots, x_n \in \{0, 1\}} f(x_1, x_2, \dots, x_n)$. At a high level, the prover and verifier follow n rounds of interaction. For each round the prover computes the univariate polynomial $p_i(x) = \sum_{\mathbf{b} \in \{0, 1\}^{n-i}} f(r_1, \dots, r_{i-1}, x, \mathbf{b})$ and sends it to the verifier. The latter checks if $p_{i-1}(r_{i-1}) = p_i(0) + p_i(1)$ (in the first round, this is $p_0(0) + p_1(1) = K$), picks a random point r_i and sends it to the prover. Finally, the verifier ends up with a claimed evaluation of f at (r_1, \dots, r_n) , the validity of which is checked by an evaluation oracle of f . The sumcheck protocol has proving complexity of $\mathcal{O}(2^n)$, proof size of $\mathcal{O}(dn)$, and soundness error of $\frac{dn}{|\mathbb{F}|}$. Throughout this paper, we consider the more general sumcheck instance of $K = \sum_{x \in \{0, 1\}^n} f(x)g(x)$.

Sumcheck Protocol. Let $f, g : \mathbb{F}^n \rightarrow \mathbb{F}$ be multi-linear polynomials of size $N = 2^n$. Assume a prover \mathcal{P} having access to f, g and a verifier \mathcal{V} having access to the evaluation oracles of f, g . Both \mathcal{P} and \mathcal{V} share a claimed value $K \in \mathbb{F}$. \mathcal{P} wants to prove that $K = \sum_{x \in \{0, 1\}^n} f(x)g(x)$.

1. \mathcal{P} computes and sends to \mathcal{V} , $p_1(z) = \sum_{\{x_2, \dots, x_n\} \in \{0, 1\}^{n-1}} f(z, x_2, \dots, x_n)g(z, x_2, \dots, x_n)$
2. \mathcal{V} ensures that $p_1(0) + p_1(1) = K$, selects a random point $r_1 \in \mathbb{F}$ and sends it to \mathcal{P} . Finally set $K \leftarrow p_1(r_1)$.
3. For all $i \in \{2, \dots, n\}$:
 - (a) \mathcal{P} computes and sends to \mathcal{V} the polynomial $p_i(z) = \sum_{\{x_{i+1}, \dots, x_n\} \in \{0, 1\}^{n-i}} f(\mathbf{r}, z, \mathbf{x})g(\mathbf{r}, z, \mathbf{x})$, with $\mathbf{r} = (r_1, \dots, r_{i-1})$ and $\mathbf{x} = (x_{i+1}, \dots, x_n)$.
 - (b) \mathcal{V} ensures that $p_i(0) + p_i(1) = K$, selects a random point $r_i \in \mathbb{F}$ and sends it to \mathcal{P} . Finally set $K \leftarrow p_i(r_i)$.
4. \mathcal{P} : Computes $y_g = g(r_1, \dots, r_n), y_f = f(r_1, \dots, r_n)$ and sends them to \mathcal{V} .
5. \mathcal{V} : Check if $K = y_g \cdot y_f$. Query the evaluation oracles to validate the correctness of y_g and y_f and return 1 if all checks passed and 0 otherwise.

GKR Protocol. Initially proposed by Goldwasser et al. [34], *GKR* is an interactive proof system for evaluating a layered arithmetic circuit \mathcal{C} of depth d . For each layer $i \in [d]$, let S_i (and $s_i = \log S_i$) be the number of gates, where S_d is the input size and S_0 output size. Similarly, we denote with $\mathbf{V}_i \in \mathbb{F}^{S_i}$ the vector that consists of the outputs of each gate of the i -th level and $f_i : \mathbb{F}^{s_i} \rightarrow \mathbb{F}$ its multi-linear extension. Finally, we can describe the wiring pattern of every layer i using the functions $add_i, mul_i : \{0, 1\}^{s_i+2s_{i+1}} \rightarrow \{0, 1\}$ such that $add_i(z, x, y) = 1$ ($mul_i(z, x, y) = 1$ resp.) if the addition (multiplication resp.) gate of the circuit with label z takes as input the output of the gates with labels x and y in the $(i+1)$ -th layer.

Given $(\mathcal{C}, \mathbf{V}_0, \mathbf{V}_d)$, the verifier selects a random point $r_0 \in \mathbb{F}^{s_0}$, computes $f_0(r_0)$, and sends r_0 to the prover. The prover and the verifier interact over the sumcheck instance:

$$f_0(r_0) = \sum_{(x, y) \in \{0, 1\}^{2s_1}} \left(f_{add_0}(r_0, x, y)(f_1(x) + f_1(y)) + f_{mul_0}(r_0, x, y)f_1(x)f_1(y) \right)$$

At the end of this sumcheck instance, verifier receives $f_1(r'_1), f_1(r''_1), f_{add_0}(r_0, r'_1, r''_1), f_{mul_0}(r_0, r'_1, r''_1)$. For the last two values, it locally checks their validity. However, this is not possible for $f_1(r'_1), f_1(r''_1)$ as it needs to evaluate the circuit. To overcome that issue, the verifier reduces these claims into one, denoted with $f_1(r_1)$, and interacts over a similar sumcheck instance but for the next layer.

We repeat the same process until the d -th layer. At the end, the verifier receives $f_d(r'_d), f_d(r''_d), f_{add_d}(r_{d-1}, r'_d, r''_d), f_{mul_d}(r_{d-1}, r'_d, r''_d)$, and can verify their validity. The resulting interactive proof has proof size $O(d \log |\mathcal{C}|)$, and optimal prover complexity of $O(|\mathcal{C}|)$ [10, 66]. Additionally, if \mathcal{C} has a sufficiently “regular” wiring pattern [67, 68, 28, 8, 69], it has verification cost of $O(\text{poly}(d, \log |\mathcal{C}|) + |V_0| + |V_d|)$.

The GKR Protocol. Let \mathcal{C} be a d -layered data-parallel arithmetic circuit. We denote with $\mathbf{V}_d \in \mathbb{F}^{S_d}$ the input values of \mathcal{C} , $\mathbf{V}_0 \in \mathbb{F}^{S_0}$ its output values (held by both the prover and verifier) and $\mathbf{V}_i \in \mathbb{F}^{S_i}$ the output values of the i -th layer of \mathcal{C} . Furthermore, we denote with $f_i : \mathbb{F}^{S_i} \rightarrow \mathbb{F}$ the multi-linear extension of \mathbf{V}_i . \mathcal{P} proves that $\mathcal{C}(\mathbf{V}_d) = \mathbf{V}_0$ by interacting with \mathcal{V} over the following protocol:

1. \mathcal{V} : Select a random point $\mathbf{r}^{(0)} \in \mathbb{F}^{s_0}$ and send it to \mathcal{P} .
2. \mathcal{P} : Evaluates $f_0(\mathbf{r}^{(0)})$ and sends $f_0(\mathbf{r}^{(0)})$ to \mathcal{V} .
3. \mathcal{P} - \mathcal{V} : Interact following the sumcheck protocol for proving the correctness of the output layer:

$$f_d(\mathbf{r}^{(0)}) = \sum_{x, y \in \{0, 1\}^{2s_1}} f_{add}^i(\mathbf{r}^{(0)}, x, y)(f_1(x) + f_1(y)) + f_{mul}^i(\mathbf{r}^{(0)}, x, y)(f_1(x)f_1(y))$$

4. \mathcal{P} : At the end of the protocol, send $f_1(\mathbf{r}_1^{(1)}), f_1(\mathbf{r}_2^{(1)})$ to \mathcal{V} .
5. \mathcal{V} : Evaluate $u_{add} = f_{add}^i(\mathbf{r}^{(0)}, \mathbf{r}_1^{(1)}, \mathbf{r}_2^{(1)})$ $u_{mul} = f_{mul}^i(\mathbf{r}^{(0)}, \mathbf{r}_1^{(1)}, \mathbf{r}_2^{(1)})$, and check if the last round of sumcheck equals to $u_{add} \cdot (f_1(\mathbf{r}_1^{(1)}) + f_1(\mathbf{r}_2^{(1)})) + u_{mul} \cdot f_1(\mathbf{r}_1^{(1)}) \cdot f_1(\mathbf{r}_2^{(1)})$.
6. For $i = 2, \dots, d$ do:

- (a) \mathcal{V} : Randomly selects $a, b \in \mathbb{F}$ and sends them to \mathcal{P} .
- (b) \mathcal{P} - \mathcal{V} : Interact following the sumcheck protocol for proving the correctness of the $(i-1)$ -th layer:

$$a f_{i-1}(\mathbf{r}_1^{(i-1)}) + b f_{i-1}(\mathbf{r}_2^{(i-1)}) = \sum_{x \in \{0, 1\}^{s_i}} F_{add}(x, y)(f_i(x) + f_i(y)) + F_{mul}(x, y)(f_i(x)f_i(y))$$

Where $F_{add}(x, y) = a \cdot f_{add_i}(\mathbf{r}_1^{(i-1)}, x, y) + b \cdot f_{add_i}(\mathbf{r}_2^{(i-1)}, x, y)$, $F_{mul}(x, y) = a \cdot f_{mul_i}(\mathbf{r}_1^{(i-1)}, x, y) + b \cdot f_{mul_i}(\mathbf{r}_2^{(i-1)}, x, y)$

- (c) \mathcal{P} : At the end of the protocol, send $f_i(\mathbf{r}_1^{(i)}), f_i(\mathbf{r}_2^{(i)})$ to the verifier.
- (d) \mathcal{V} : Compute the evaluations of the wiring predicates and validate the last round of the sumcheck as in step (5).

7. \mathcal{V} : Use f_0, f_d to validate the correctness of $f_d(\mathbf{r}_1^{(d)}), f_d(\mathbf{r}_2^{(d)}), f_d(\mathbf{r}_0^{(0)})$.

2.2 Polynomial Commitments

A *polynomial commitment* scheme (PC) enables a prover to commit to an n -variate polynomial of degree d and later generate a proof that it correctly evaluated the committed polynomial at a random point. A (binding) PC scheme consists of four probabilistic polynomial time algorithms ($Gen, Commit, Eval, Verify$). At a high level, Gen , generates some public parameters, $Commit$, takes as input the public parameters, a polynomial f , and generates a succinct commitment C_f of

the polynomial. Finally, *Eval* takes as input the committed polynomial f , an evaluation point \mathbf{x} , and outputs the evaluation y and an evaluation proof π . The verifier invokes *Verify* using π , \mathbf{x} and y to validate that $y = f(\mathbf{x})$. A PC is *knowledge sound* if, for any Probabilistic Polynomial Time (PPT) adversary that generates an accepting evaluation proof, there exists a PPT extractor that extracts the committed polynomial f , such that the probability that $f(\mathbf{x}) \neq y$ is negligible with respect to the security parameter λ . A PC scheme is *complete* if the verifier always accepts the proof for a correctly evaluated point. Finally, a PC scheme is *zero-knowledge* if a verifier, given a valid proof, a point, and the evaluation of the committed polynomial at that point, learns nothing more other than the validity of the proof. We give a formal definition of a PC scheme in Appendix 2.2.

Definition 2.2 (Polynomial Commitment) *A polynomial commitment (PC) scheme is a tuple of four algorithms (Gen, Commit, Eval, Verify) defined as follows:*

- $\mathbf{pk}, \mathbf{vk} \leftarrow PC.Gen(1^\lambda, l, d)$. Given the security parameter λ , number of variables l and degree d , generates the public parameters \mathbf{pk}, \mathbf{vk} .
- $C_f \leftarrow PC.Commit(\mathbf{pk}, f, r)$. Given \mathbf{pk} and a l -variate polynomial f of degree d and randomness $r \in \mathbb{F}$, returns a polynomial commitment C_f .
- $y, \pi \leftarrow PC.Eval(\mathbf{pk}, f, \mathbf{x})$. Given \mathbf{pk} , C_f , f and a point $x \in \mathbb{F}^l$, returns $y = f(\mathbf{x})$ and a proof π that $f(\mathbf{x}) = y$.
- $0, 1 \leftarrow PC.Verify(\mathbf{vk}, \pi, C_f, \mathbf{x}, y)$. Given \mathbf{vk} , the commitment C_f , the proof π , the y and \mathbf{x} , returns 1 if $f(\mathbf{x}) = y$, 0 otherwise.

For a PC scheme the following must hold:

- **Completeness.** A polynomial commitment scheme is complete, if for any λ , and number of variables l , we have:

$$\Pr \left(\begin{array}{l} \mathbf{pk}, \mathbf{vk} \leftarrow PC.Gen(1^\lambda, l, d), \\ C_f \leftarrow PC.Commit(\mathbf{pk}, f, r) \\ y, \pi \leftarrow PC.Eval(\mathbf{pk}, f, \mathbf{x}); \\ PC.Verify(\mathbf{vk}, C, \pi, y, \mathbf{x}) = 1 \end{array} \right) = 1$$

- **Knowledge Soundness.** A polynomial commitment scheme is knowledge sound, if for any λ, l, d and any probabilistic polynomial time adversaries \mathcal{A}_{PC} there exists an extractor \mathcal{E}_{PC} that has access to the random tape of \mathcal{A}_{PC} such that:

$$\Pr \left(\begin{array}{l} \mathbf{pk}, \mathbf{vk} \leftarrow PC.Gen(1^\lambda, l, d), \\ C_f, y, \pi, \mathbf{x} \leftarrow \mathcal{A}(\mathbf{pk}, \mathbf{vk}) : \\ f, r \leftarrow \mathcal{E}_{PC}(\mathbf{pk}, \mathbf{vk}) \wedge \\ PC.Verify(\mathbf{vk}, C_f, \pi, y, \mathbf{x}) = 1 \wedge \\ (f(\mathbf{x}) \neq y \vee C_f \neq PC.Commit(\mathbf{pk}, f, r)) \end{array} \right) \leq \text{negl}(\lambda)$$

- **Zero-Knowledge.** An additional property that some polynomial commitments carry is the hiding property [27]. Although this property is not always necessary, we use it to make our SNARK zero-knowledge. Having established that, a polynomial commitment scheme is Zero Knowledge if for any λ, l, d , adversary \mathcal{A} and simulator \mathcal{S} we have:

$$\Pr (Real_{\mathcal{A}, f}(1^\lambda) = 1) \approx \Pr (Ideal_{\mathcal{A}, \mathcal{S}}(1^\lambda) = 1)$$

Real_{A,f}(1^λ):

1. $\mathbf{pk}, \mathbf{vk} \leftarrow PC.Gen(1^\lambda, l, d)$
2. $C_f \leftarrow PC.Commit(\mathbf{pk}, f, r_f)$
3. $n \leftarrow \mathcal{A}(1^\lambda, \mathbf{pk}, \mathbf{vk}, C_f)$
4. For each step $j \in \{2, 3, \dots, n\}$:
 - (a) $\mathbf{x}_i \leftarrow \mathcal{A}(1^\lambda, C_f, y_1, \dots, y_{j-1}, \pi_1, \dots, \pi_{j-1}, pp)$
 - (b) $y_j, \pi_j \leftarrow PC.Eval(pp, f, \mathbf{x}_i)$
5. Output $b \leftarrow \mathcal{A}(1^\lambda, C_f, y_1, \dots, y_n, \pi_1, \dots, \pi_n, pp)$.

Ideal_{A,S}(1^λ):

1. $(C_f, \mathbf{pk}, \mathbf{vk}, t) \leftarrow \mathcal{S}(1^\lambda, l)$
2. $n \leftarrow \mathcal{A}(1^\lambda, pp, C_f)$
3. For each step $j \in \{2, 3, \dots, n\}$:
 - (a) $\mathbf{x}_i \leftarrow \mathcal{A}(1^\lambda, C_f, y_1, \dots, y_{j-1}, \pi_1, \dots, \pi_{j-1}, pp)$
 - (b) $y_j, \pi_j \leftarrow \mathcal{S}(pp, f, \mathbf{x}_i)$
4. Output $b \leftarrow \mathcal{A}(1^\lambda, C_f, y_1, \dots, y_n, \pi_1, \dots, \pi_n, pp)$.

2.3 Argument Systems

Argument Systems. *Argument systems* [1] are similar to interactive proofs but their soundness only holds against computationally-bounded PPT adversaries. An interactive argument system for an NP language \mathcal{L} is a protocol between a verifier and a prover, in which given common input \mathbf{x} , the prover tries to convince the verifier that exists a witness \mathbf{w} , such that $(\mathbf{x}; \mathbf{w}) \in \mathcal{R}_{\mathcal{L}}$. Such a system is *knowledge sound* if, whenever a verifier accepts for a statement \mathbf{x} , there exists an efficient *extractor* program that can produce a valid witness with overwhelming (in the security parameter) probability. An argument system is *zero-knowledge* if the verifier learns nothing from interacting with the prover besides $\mathbf{x} \in \mathcal{L}$. This is captured by requiring the existence of an efficient *simulator* that interacts with the verifier without having \mathbf{w} in an indistinguishable manner. When proof size and verification time are poly-logarithmically bounded to the witness length, the argument is *succinct* and if there is no (two-way) interaction between prover and verifier, it is *non-interactive*. When both are satisfied the construction is a *zkSNARK* (e.g., [70, 3, 28, 71]). Formally:

Definition 2.3 (Zero-Knowledge Arguments of Knowledge) For any NP relation \mathcal{R} , a tuple of probabilistic polynomial time (PPT) algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a Zero-Knowledge Argument of Knowledge if it satisfies the following:

- **Completeness.** For any security parameter λ , $(pk, vk) \leftarrow \mathcal{G}(1^\lambda)$ and $(x, w) \in R$, $\Pr(\mathcal{V}(vk, x, t) = \text{accept}) = 1$, where $t = \langle \mathcal{P}(pk, w), \mathcal{V}(vk) \rangle(x)$.
- **Knowledge Soundness.** For every PPT prover \mathcal{P}^* there exists a PPT extractor \mathcal{E} that have access to \mathcal{P}^* 's random tape, such that for any security parameter λ , $(pk, vk) \leftarrow G(1^\lambda)$ and x it holds that:

$$\Pr \left(\begin{array}{l} t = \langle \mathcal{P}^*(pk), \mathcal{V}(vk) \rangle(x), \\ \mathcal{V}(vk, t, x) = \text{accept} \wedge \\ w \leftarrow \mathcal{E}(pk, x) \wedge \\ (x, w) \notin \mathcal{R} \end{array} \right) \leq \text{negl}(\lambda)$$

- **Zero-Knowledge** There exists a PPT simulator \mathcal{S} such that for any PPT adversary \mathcal{A} , $(x, w) \in \mathcal{R}$ and auxiliary input $z \in \{0, 1\}^*$, it holds that:

$$\text{View}(t) \approx \text{View}(t^*)$$

Where $t = \langle \mathcal{P}(pk, w), \mathcal{A}(vk) \rangle(x)$ and $t^* = \langle \mathcal{S}(pk, z), \mathcal{A}(vk) \rangle(x)$ are accepting transcripts and $\text{View}(\cdot)$ denotes the distribution of the transcript.

We can construct a zero-knowledge argument of knowledge from the GKR protocol using PC schemes [28], by (i) committing to the multi-linear extension of the witness $f_{\mathbf{w}}$, (ii) using the GKR protocol to prove that $\mathcal{C}(\mathbf{x}, \mathbf{w}) = 1$, where \mathcal{C} is a layered arithmetic circuit, and (iii) opening $f_{\mathbf{w}}$ at the challenge point derived at the last round of the protocol. Moreover, following the observations of [67, 68, 28, 8, 69], if the circuit \mathcal{C} is log-space uniform, then the previous scheme is also succinct with proof size $\mathcal{O}(d \log |\mathcal{C}|)$ and verification time $\text{polylog}|\mathcal{C}| + |\mathbf{x}|$.

2.4 Decision Tree and Random Forest Training

Without loss of generality, we focus on training binary decision trees. Let $D \in \mathbb{R}^{n \times d}$ be a dataset that consists of n data points with d features, and $y \in \{0, 1\}^n$ be the class assigned to each point. We consider the standard training algorithm [72, 73, 74] that relies on the following divide-and-conquer algorithm. First, we scan D and find the feature i and value s that split the dataset into D_L, D_R , in such a way that each partitioned dataset is as “pure” as possible (i.e.,—almost—all its instances have the same label), where $\forall p \in D_L, p_i < s$ and $\forall p \in D_R, p_i \geq s$. Then, create a node storing i, s , and continue recursively the same process for its left child, using D_L , and its right child, using D_R (without considering the i -th feature). The process ends if D_L and D_R contain a single point, or all of their points have the same label or a maximum depth is reached.

What remains is to compute the best split. In this work, we consider a histogram-based strategy [75, 76, 77, 78], in which we bucket every value of the dataset into B bins (e.g., $B = 128$), and use the “discretized” dataset $D' \in [B]^{n \times d}$ to compute a histogram $H_{i,c} \in \mathbb{N}^B$, for each feature $i \in [d]$ and class $c \in \{0, 1\}$, defined as $H_{i,c}[j] = \sum_{k \in [n]} I(p_{k,j} = j \wedge y_k = c)$. Then, for every attribute $i \in [d]$ and each bin $j \in [B]$, we compute its split quality $\epsilon_{i,j}$ by using an impurity function. Here, we use the Gini [72] function defined as $Gini(P_0, P_1) = 1 - \sum_{c \in \{0,1\}} P_c^2$ where P_c is the probability a point in D belongs to the c -th class. Based on that, we compute the probabilities $P_c = \frac{1}{n_1} \sum_{k \in \{0, \dots, j-1\}} H_{i,c}[k]$ and $P'_c = \frac{1}{n_2} \sum_{k \in \{j, \dots, B\}} H_{i,c}[k]$, where n_1 is the number of points of the left partition and n_2 of the right one, and set $\epsilon_{i,j} = \frac{n_1}{n} Gini(P_0, P_1) + \frac{n_2}{n} Gini(P'_0, P'_1)$. Finally, select feature i and value j with the smallest $\epsilon_{i,j}$.

A random forest classifier [44], consists of K decision trees. To train a random forest we first perform a *Bagging Step* to generate K new datasets $D_i \in \mathbb{R}^{n \times d}$ by picking n random points from D with replacement. Next, we invoke a slightly different version of the training algorithm for each D_i , which performs *feature bagging*, e.g., split a node using a random subset of features. We provide the training and inference algorithms in Appedix C.1, denoted as TRAIN and PREDICT respectively.

3 Sparrow: A Space-Efficient zkSNARK

Here we present our space-efficient zkSNARK SPARROW for layered data-parallel arithmetic circuits. In Section 3.1, we introduce a space-efficient sumcheck with improved prover time and in Section 3.2 we introduce our zkSNARK based on a novel space-efficient variant of GKR and a space-efficient PC scheme.

3.1 Our Space-Efficient Sumcheck

In this section, we develop a new space-efficient sumcheck protocol for the instance $K = \sum_{x \in \{0,1\}^n} f(x)g(x)$, where $N = 2^n$ and $f, g : \mathbb{F}^n \rightarrow \mathbb{F}$, are the multi-linear extensions of the vectors $\mathbf{A}_f, \mathbf{A}_g \in \mathbb{F}^N$. Our construction operates in the streaming model [19, 26], in which the prover accesses vectors $\mathbf{A}_f, \mathbf{A}_g$ via corresponding read-only streaming oracles $S(\mathbf{A}_f), S(\mathbf{A}_g)$, and only maintains a smaller “buffer”

workspace for storage. We consider these oracles as routines that internally keep some state (e.g., the latest accessed position i), and upon invocation return the next element or vector of elements.

The “straightforward” extension of the standard sumcheck protocol in this streaming model has already been explored in prior works [26, 19, 21]; at a high level, it can be described as follows. In the i -th sumcheck round (for $i = 1, \dots, \log N$), the prover must compute the univariate polynomial $p_i(x)$ (see section 2.1). This requires first computing polynomials $f(r_1, \dots, r_{i-1}, x, b_{i+1}, \dots, b_n)$, $g(r_1, \dots, r_{i-1}, x, b_{i+1}, \dots, b_n)$, $\forall \{b_{i+1}, \dots, b_n\} \in \{0, 1\}^{n-i}$, “on-the-fly”, given access to the streaming oracles $S(\mathbf{A}_f), S(\mathbf{A}_g)$. Computing these two polynomials for each round requires $\mathcal{O}(\log N)$ space, thus the overall space-efficiency of this protocol is also $\mathcal{O}(\log N)$. However, since the prover does not have sufficient buffer space, for each of the $\log N$ rounds, it needs to scan the entire vectors $\mathbf{A}_f, \mathbf{A}_g$, hence its overall time is $\mathcal{O}(N \log N)$. In fact, it is easy to see that this approach will result in $\mathcal{O}(N \log N)$ field operations, even if we relax the prover’s space to $o(N)$! For instance, increasing the buffer size to $\mathcal{O}(\sqrt{N})$ the prover still needs to scan the vectors $\log N/2$ times and perform $\mathcal{O}(N)$ stream accesses and field operations on every scan until can store the “reduced” polynomials in its buffer and run the rest of the protocol with the standard sumcheck.

In contrast, our proposed sumcheck achieves $\mathcal{O}(N \log \log N)$ field operations, at the cost of relaxing space efficiency to $\mathcal{O}(\sqrt{N})$, which we consider a reasonable trade-off for practical purposes.

Main Idea of our Protocol. Based on the above, we observe that to have any hope of improving the proving time, we must reduce the number of times we scan the vectors. In turn, this will lead to a reduction in the number of rounds. We achieve this by reducing the number of variables of the polynomials, i.e., instead of using the multi-linear polynomials f, g directly, we replace them with “equivalent” polynomials \hat{f}, \hat{g} with fewer higher-degree variables.

The main challenge is to carefully select the number of variables and their degrees to achieve the desired asymptotics. We base our design on the following observations. First, because we set the buffer size of the prover to $\mathcal{O}(\sqrt{N})$, we only need to replace the first $(\log N)/2$ variables of f, g . That is because in the last $(\log N)/2$ rounds of the sumcheck protocol, we can directly fit the reduced multi-linear polynomials in our buffer. Second, the number of high-degree variables must be asymptotically smaller than $\log N$. Otherwise, the prover would have to scan the oracles $\mathcal{O}(\log N)$ times, resulting in $\mathcal{O}(N \log N)$ proving time. Finally, when choosing the variables in \hat{f}, \hat{g} we note that we should order them from smaller to larger degree (excluding the last $(\log N)/2$ variables of degree 1). The intuition behind this comes from the fact that: (i) to compute the polynomial p_i , for a d -degree variable x_i , we need to multiply polynomials of degree d and (ii) the number of polynomial multiplications significantly decreases in every round as we reduce the total size of \hat{f}, \hat{g} . Naturally, we wish to perform more multiplications for polynomials of smaller degrees and vice-versa.

These observations lead us to the following strategy. Replace the first $(\log N)/2$ variables with a constant number of high-degree ones, and assign their degrees in increasing order (note that the product of their degrees must be \sqrt{N}). Furthermore, we select the degrees such that only the first round dominates the proving time. In practice, we replace the first $\log N/2$ variables with only two variables, of degree $\log N$ and $\sqrt{N}/\log N$, respectively. Thus, in the first round, we need to multiply $N/\log N$ polynomials of degree $\log N$. Since polynomial multiplication requires $\mathcal{O}(\log N \log \log N)$ field operations and we need to perform $N/\log N$ such multiplications, the total complexity of the first round will be $\mathcal{O}(N \log \log N)$. In the second round, we multiply \sqrt{N} polynomials of degree $\sqrt{N}/\log N$, leading to $\mathcal{O}(\log \sqrt{N} \cdot (\sqrt{N}/\log N) \cdot \sqrt{N}) = \mathcal{O}(N)$ field operations.

Formally, we replace $f : \mathbb{F}^n \rightarrow \mathbb{F}$ with the polynomial $\hat{f} : \mathbb{F}^{2+n/2} \rightarrow \mathbb{F}$, defined as $\hat{f}(x_0, x_1, \mathbf{x}_2)$ equal to

$$\sum_{i \in \mathbb{H}_0, j \in \mathbb{H}_1, \mathbf{k} \in \{0, 1\}^{n/2}} L_i^{(0)}(x_0) L_j^{(1)}(x_1) \beta(\mathbf{x}_2, \mathbf{k}) \mathbf{A}_f [2^{n-\log n} i + 2^{n/2} j + k]$$

where \mathbb{H}_0 and \mathbb{H}_1 are multiplicative sub-groups of \mathbb{F} of size n and $2^{n-\log n}$, respectively, $L_i^{(0)}(x)$ and $L_j^{(1)}(x)$ their Lagrange polynomials, and \mathbf{k} is the bit-decomposition of k . Observe that there is a one-to-one mapping between all points in $\mathbf{x} \in \mathbb{H}_0 \times \mathbb{H}_1 \times \{0, 1\}^{n/2}$ and $\mathbf{z} \in \{0, 1\}^n$, such that $\hat{f}(\mathbf{x}) = f(\mathbf{z})$. Likewise, we define \hat{g} for g .

Having established that, we partition our protocol into two phases. In the first phase, we replace the original claim with proving that $K = \sum_{x_0, x_1, \mathbf{x}_2 \in \mathbb{H}_0 \times \mathbb{H}_1 \times \{0, 1\}^{n/2}} \hat{f}(x_0, x_1, \mathbf{x}_2) \hat{g}(x_0, x_1, \mathbf{x}_2)$. In the second phase, we “reduce” the evaluation claims of \hat{f}, \hat{g} , derived from the first phase, into evaluation claims of f, g . **Protocol 1** gives a detailed description of our sumcheck protocol.

Protocol 1: Let $f, g : \mathbb{F}^n \rightarrow \mathbb{F}$ be the multi-linear extensions of $\mathbf{A}_f, \mathbf{A}_g$ and $S(\mathbf{A}_f), S(\mathbf{A}_g)$ their streaming oracles. We want to prove that $K = \sum_{x \in \{0, 1\}^n} f(x)g(x)$. Our protocol takes as input $S(\mathbf{A}_f), S(\mathbf{A}_g)$, n and public parameters of a PC scheme.

• **Initialization:** Set $\tilde{\mathbf{A}}_f \leftarrow \{0\}^{2^{n/2}}, \tilde{\mathbf{A}}_g \leftarrow \{0\}^{2^{n/2}}, p_0(x) \leftarrow 0, p_1(x) \leftarrow 0, \mathbf{F} \leftarrow \{0\}^{2^{n/2}}, \mathbf{G} \leftarrow \{0\}^{2^{n/2}}$.

• **Phase 1:** For \hat{f}, \hat{g} the equivalent polynomials of f, g , prove that $K = \sum_{\substack{x_0, x_1, \mathbf{x}_2 \\ \in \mathbb{H}_0 \times \mathbb{H}_1 \times \{0, 1\}^{n/2}}} \hat{f}(x_0, x_1, \mathbf{x}_2) \hat{g}(x_0, x_1, \mathbf{x}_2)$.

– **First round** (over the n -degree variable x_0):

1. \mathcal{P} : For $i = 1, \dots, 2^{n-\log n}$: Read the next n elements of $S(\mathbf{A}_f), S(\mathbf{A}_g)$ and store them in $\tilde{\mathbf{A}}_f, \tilde{\mathbf{A}}_g$. Set $p_f(x) = \sum_{j \in [n]} L_j^{(0)}(x) \tilde{\mathbf{A}}_f[j]$ (and likewise $p_g(x)$ using $\tilde{\mathbf{A}}_g$) and compute $p_0(x) \leftarrow p_0(x) + p_f(x)p_g(x)$ (via FFT).
2. \mathcal{V} : Receive $p_0(x)$, check if $K = \sum_{i \in [2n]} p_0(\omega_1^i)$, pick a random point r_0 , compute $p_0(r_0)$ and send r_0 to \mathcal{P} .

– **Second round** (over the $2^{n-\log n}$ -degree variable x_1):

1. \mathcal{P} : For $i = 1, \dots, 2^{n/2}$: Read the next $2^{n/2}$ elements of $S(\mathbf{A}_f), S(\mathbf{A}_g)$, store them in $\tilde{\mathbf{A}}_f, \tilde{\mathbf{A}}_g$ and compute the polynomial $p_f(x) = \hat{f}(r_0, x_1, \mathbf{i})$ defined as $\sum_{j \in [2^{n-\log n}]} L_j^1(x) c_j$, where $c_j = \sum_{k \in [n]} \mathbf{y}_0[k] \tilde{\mathbf{A}}_f[jn+k]$ and $\mathbf{y}_0 = (L_1^0(r_0), \dots, L_n^0(r_0))$. Likewise compute $p_g(x)$ using $\tilde{\mathbf{A}}_g$. Finally compute $p_1(x) \leftarrow p_1(x) + p_f(x)p_g(x)$ (via FFT).
2. \mathcal{V} : Delegates its checks to \mathcal{P} with the following protocol:
 - (a) \mathcal{P} : Parse $p_1(x)$ as $\sum_{i \in [2^{n/2-\log n+1}]} c_i \tilde{L}_i(x)$, set $l \leftarrow n/2 - \log n$, $\mathbf{c} = (c_1, \dots, c_{2^{l+1}})$ and commit to $p_1(x)$ by committing to $f_{\mathbf{c}}$ using the PC scheme. Send C_{p_1} to \mathcal{V} and receive r_1 .
 - (b) \mathcal{P} : interacts with \mathcal{V} following the standard sumcheck protocol to prove that $p_0(r_0) = \sum_{i \in [2^l]} p_1(\omega^i)$, using the instance $p_0(r_0) = \sum_{x \in \{0, 1\}^l} f_{\mathbf{c}}(x)$. \mathcal{P} proves the validity of the claimed evaluation $y_{p_1}^{(1)}$ using the PC scheme.
 - (c) \mathcal{P} : Compute $\mathbf{w}_1 = (1, \omega, \dots, \omega^{2^{l+1}-1}), \mathbf{w}_2 = (r_1^{-1}, (r_1 - \omega)^{-1}, \dots, (r_1 - \omega^{2^{l+1}-1})^{-1}), \mathbf{w}_3 = \mathbf{w}_2^{-1}$, commit to $f_{\mathbf{w}_2}$ using the PC scheme, send $C_{\mathbf{w}_2}$ to \mathcal{V} and receive a random challenge \mathbf{r}' . Note that $\tilde{L}_i(r_1) = K \mathbf{w}_1[i] \mathbf{w}_2[i]$, where $K = (r_1^{2^{l+1}} - 1)/2^{l+1}$.
 - (d) \mathcal{P} interacts with \mathcal{V} following the standard sumcheck protocol to prove the correct computation of p_1 at \mathbf{r}_1 , using the instance $p_1(r_1) = K \sum_{x \in \{0, 1\}^{l+1}} p_1(x) f_{\mathbf{w}_1}(x) f_{\mathbf{w}_2}(x)$. \mathcal{P} proves the validity of the claimed evaluation $y_{p_1}^{(2)}, y_{\mathbf{w}_2}^{(1)}$ using the PC scheme. \mathcal{V} locally validates $y_{\mathbf{w}_1}$.
 - (e) \mathcal{P} interacts with \mathcal{V} with the sumcheck protocol to prove “well-formedness” of \mathbf{w}_2 , using the instance $1 = \sum_{x \in \{0, 1\}^{l+1}} \beta(x, \mathbf{r}') f_{\mathbf{w}_2}(x) f_{\mathbf{w}_3}(x)$. \mathcal{P} proves the validity of the evaluation $y_{\mathbf{w}_2}^{(2)}$ via the PC scheme. \mathcal{V} locally checks $y_{\mathbf{w}_3}$.

– **Remaining rounds** (over the 1-degree variables $x_2, \dots, x_{n/2+2}$):

1. \mathcal{P} : For $i = 1, \dots, 2^{n/2}$: Read the next $2^{n/2}$ elements of $S(\mathbf{A}_f), S(\mathbf{A}_g)$, store them in $\tilde{\mathbf{A}}_f, \tilde{\mathbf{A}}_g$ and compute $\mathbf{F}[i] = \hat{f}(r_0, r_1, \mathbf{i}) = \sum_{j \in [2^{n/2}]} (\mathbf{y}_0 \otimes \mathbf{y}_1)[j] \tilde{\mathbf{A}}_f[j]$ (and similarly compute $\mathbf{G}[i]$ using $\tilde{\mathbf{A}}_g$), where $\mathbf{y}_1 = (L_1^1(r_1), \dots, L_{2^l}^1(r_1))$.
2. \mathcal{P} : interacts with \mathcal{V} following the standard sumcheck protocol to prove that $p_1(r_1)$ equals to $\sum_{x \in \{0,1\}^{n/2}} \hat{f}(r_0, r_1, x) \hat{g}(r_0, r_1, x)$. Finally, \mathcal{V} ends up with claimed evaluations $y_{\hat{f}}, y_{\hat{g}}$ at (r_0, r_1, \mathbf{r}_2) .

• **Phase 2:** Reduce the evaluation claims of \hat{f}, \hat{g} into claims of f, g .

1. \mathcal{P} : For $i = 1, \dots, 2^{n/2}$: Read $2^{n/2}$ elements from $S(\mathbf{A}_f), S(\mathbf{A}_g)$ and store them in $\tilde{\mathbf{A}}_f, \tilde{\mathbf{A}}_g$. Update \mathbf{F} (and similarly \mathbf{G}) by computing $\mathbf{F}[j] = \mathbf{F}[j] + \beta(\mathbf{i}, \mathbf{r}_2) \mathbf{A}_f[j]$ for each $j \in [2^{n/2}]$. At the end of the iteration, \mathcal{P} holds $f(x, \mathbf{r}_2), g(x, \mathbf{r}_2)$.
2. \mathcal{P} receives $a_1, a_2 \in \mathbb{F}$ from \mathcal{V} and interacts with it following the sumcheck protocol for the instance $a_1 y_{\hat{f}} + a_2 y_{\hat{g}} = \sum_{x \in \{0,1\}^{n/2}} f_{\mathbf{y}}(x) (a_1 f(x, \mathbf{r}_2) + a_2 g(x, \mathbf{r}_2))$. \mathcal{P} sends $y_{\mathbf{y}_0}, y_{\mathbf{y}_1}, y_f, y_g$ and \mathcal{V} validates the last round of the sumcheck protocol.
3. \mathcal{V} checks the correctness of $y_{\mathbf{y}_0}$ locally and for $y_{\mathbf{y}_1}$, interacts with \mathcal{P} following steps 2.c-2.e (but in step 2.d replaces $p_1(x)$ with the multi-linear polynomial s.t $l(x) = \beta(x, r_{\mathbf{y}_1})$).

Phase 1: Sumcheck over \hat{f}, \hat{g} . After initializing the necessary data-structures in the buffer space, we proceed as follows.

First round: In this round, we compute and send the univariate polynomial $p_0(x)$ of the n -degree variable (step 1 of **First round**). Specifically, we make a streaming pass over the vectors reading n elements at a time, form the polynomials $p_f(x) = \hat{f}(x, h_2, \mathbf{b}), h_2 \in \mathbb{H}_1, \mathbf{b} \in \{0, 1\}^{n/2}$ and $p_g(x)$ respectively, which we multiply (via FFT¹), to update the polynomial $p_0(x)$ by setting $p_0(x) \leftarrow p_0(x) + p_f(x)p_g(x)$. Finally, send $p_0(x)$ to the verifier, which checks if $K = \sum_{\omega \in \mathbb{H}_0} p_0(\omega)$ and replies with a random point $r_0 \in \mathbb{F}$ (step 2 of **First round**).

Second round: Similarly with the previous round, we make a streaming pass over the vectors. This time we read chunks of $2^{n/2}$ elements and use them to evaluate the reduced polynomials $p_f(x) = \hat{f}(r_0, x, \mathbf{b}), p_g(x) = \hat{g}(r_0, x, \mathbf{b})$ for $\mathbf{b} \in \{0, 1\}^{n/2}$ which we multiply to compute $p_1(x)$ (step 1 of **Second round**). At this point, recall that after receiving $p_1(x)$, the verifier has to (i) evaluate $p_1(x)$ at a random point r_1 and (ii) check if $p_0(r_0) = \sum_{\omega \in \mathbb{H}_1} p_1(\omega)$. Unfortunately, $p_1(x)$ is a $2^{n/2 - \log n + 1}$ -degree polynomial. Therefore, to preserve the succinctness of our protocol, we delegate these checks to the prover (step 2 of **Second round**).

Because $p_1(x)$ is univariate, we could directly apply the univariate sumcheck protocol [79, 32] to prove (i) and (ii). However, this would either increase the overall proof size by a logarithmic factor [32], or require additional trusted public parameters for univariate polynomial commitments [79]. (We also note that using [79, 32] would give a multiplicative logarithmic overhead for the proving time of this step, whereas our approach achieves linear time.) Instead, we propose an approach that solely relies on multi-linear polynomials and, thus, is directly compatible with the remaining of our protocol. At a high level, we first commit multi-linear extension of the coefficients of $p_1(x)$ using a multi-variate PC [80], map the checks (i),(ii) as sumcheck instances, and invoke the standard sumcheck protocol to prove their correctness.

More precisely, we first commit to the polynomial $p_1(x) = \sum_{i \in [2^{l+1}]} c_i \tilde{L}_i^1(x)$, where $l = n/2 - \log n$, by committing to the multi-linear extension $f_{\mathbf{c}} : \mathbb{F}^{l+1} \rightarrow \mathbb{F}$ of $\mathbf{c} = (c_1, \dots, c_{2^{l+1}})$, using a multi-variate PC scheme [80] (line 14). Then, the prover sends the commitment to the verifier and receives r_1 (step 2.a).

¹In fact, since p_f, p_g are Lagrange polynomials, we first perform IFFT to compute their coefficients and then their product using the FFT-based algorithm. Every such operation requires $\mathcal{O}(n \log n)$ field operations and $\mathcal{O}(1)$ field inversions. Because the elements we invert are the same for all multiplications, we do not need to perform inversions on every multiplication but only once (e.g., when doing the first multiplication). See Appendix A.1 for more details.

For (i), we must prove that the sum of $f_{\mathbf{c}}(x)$ at all elements of \mathbb{H}_1 equals $p_0(r_0)$. Without loss of generality, we assume there is a one-to-one mapping between $\mathbf{x} \in \{0, 1\}^l$ and $i \in [2^l]$, s.t. $f_{\mathbf{c}}(\mathbf{x}) = p_1(\omega^i)$. Thus, we use the standard sumcheck protocol to prove that $p_0(r_0) = \sum_{x \in \{0, 1\}^l} f_{\mathbf{c}}(x)$ (line 14). Finally, we end up with a claim for the evaluation of $f_{\mathbf{c}}$ at a random point and prove its validity via the PC scheme (step 2.b).

For (ii) (steps 2.c-2.e), we observe that $p_1(r_1) = \sum_{i \in [2^{l+1}]} \tilde{L}_i^{(1)}(r_1) c_i = K \sum_{i \in [2^{l+1}]} \omega^i (r_1 - \omega^i)^{-1} c_i$, for $K = (r_1^{2^{l+1}} - 1) 2^{-(l+1)}$ constant. By setting $\mathbf{w}_1 = (1, \omega, \dots, \omega^{2^{l+1}-1})$ and $\mathbf{w}_2 = (r_1^{-1}, \dots, (r_1 - \omega^{2^{l+1}-1})^{-1})$, we can express $p_1(r_1)$ as $p_1(r_1) = K \sum_{\mathbf{x} \in \{0, 1\}^{l+1}} f_{\mathbf{w}_1}(\mathbf{x}) f_{\mathbf{w}_2}(\mathbf{x}) f_{\mathbf{c}}(\mathbf{x})$. To prove this sumcheck instance, we first commit to the polynomial $f_{\mathbf{w}_2}$ and invoke the standard sumcheck protocol. Finally, we end up with three claims y_{w_1}, y_{w_2} and y_c at a random point. We prove the validity of y_{w_2} and y_c with the PC scheme. For y_{w_1} , we observe that $f_{\mathbf{w}_1}$ was a multiplicative structure; so we can verify the validity of y_{w_1} in $\mathcal{O}(l)$ time. What remains is to prove the “well-formedness” of $f_{\mathbf{w}_2}$, namely, for all $i \in [2^{l+1}]$, $\mathbf{w}_2[i](r_1 - \omega^i) = 1$. We do so by translating this check to a sumcheck instance $1 = \sum_{x \in \{0, 1\}^{l+1}} \beta(x, \mathbf{r}') f_{\mathbf{w}_3}(x) f_{\mathbf{w}_2}(x)$, for a randomly selected point \mathbf{r}' and $f_{\mathbf{w}_3}$ the multi-linear extension of $\mathbf{w}_3 = (r_1, \dots, (r_1 - \omega^{2^{l+1}-1}))$. Because $f_{\mathbf{w}_3}$ has a multiplicative structure, the prover does not need to compute its commitment.

Rest of the rounds: At this point, we have the multi-linear polynomials $\hat{f}(r_0, r_1, \mathbf{x}_2), \hat{g}(r_0, r_1, \mathbf{x}_2)$, where r_0, r_1 are the random challenges of the first two rounds. Because the size of these polynomials is $\mathcal{O}(\sqrt{N})$, we can compute and store locally all their boolean hypercube evaluations, via a single pass over $\mathbf{A}_g, \mathbf{A}_f$ (step 1 of **Remaining rounds**). We then invoke the standard sumcheck over the stored evaluations \mathbf{F}, \mathbf{G} to complete the sumcheck protocol over the polynomials \hat{f}, \hat{g} (step 2 of **Remaining rounds**).

Phase 2: Reducing evaluations of \hat{f}, \hat{g} to f, g . After the completion of the first phase, we end up with $y_{\hat{f}}, y_{\hat{g}}$, the evaluations of \hat{f}, \hat{g} at (r_0, r_1, \mathbf{r}_2) . Unfortunately, the verifier can only validate evaluations of the multi-linear polynomials f, g . To reduce these claims into evaluations of f, g at a random point, we observe that $y_{\hat{f}}$ (resp. $y_{\hat{g}}$), can be re-written as $y_{\hat{f}} = \sum_{x \in \{0, 1\}^{n/2}} f_y(x) f(x, \mathbf{r}_2)$, where f_y is the the multi-linear extension of the vector $\mathbf{y}_0 \otimes \mathbf{y}_1$, and for $i \in \{0, 1\}$, $\mathbf{y}_i = ((L_1^i(r_i), \dots, L_{|\mathbb{H}_i|}^i(r_i)))$ is the vector of evaluations of all Lagrange coefficients of \mathbb{H}_i at r_i . Because the polynomials involved have size \sqrt{N} , we can prove the latter sums, again using the standard sumcheck. In more detail, we first perform a single pass over $\mathbf{A}_f, \mathbf{A}_g$ via the streaming oracles, computing the hypercube evaluations of $f(x, \mathbf{r}_2), g(x, \mathbf{r}_2)$ and storing them into our buffer space (step 1). Then, the prover receives two challenge points a_1, a_2 , batches the two sumchecks using the standard technique of [81], and uses the stored evaluations to generate a sumcheck proof for the batched instance (step 2). Thus, the verifier gets claims of f, g for the same evaluation point.

It remains to prove the correct evaluation of $f_{\mathbf{y}}$ at point \mathbf{r}_3 . As $f_{\mathbf{y}}(x) = f_{\mathbf{y}_1}(x_1) f_{\mathbf{y}_2}(x_2)$, where $x = (x_1, x_2)$, it is enough to check $y_{\mathbf{y}_0 \otimes \mathbf{y}_1} = f_{\mathbf{y}_1}(\mathbf{r}_{3,1}) f_{\mathbf{y}_2}(\mathbf{r}_{3,2}) = y_{\mathbf{y}_0} y_{\mathbf{y}_1}$. Although the correctness of $y_{\mathbf{y}_0}$ can be checked in time $\mathcal{O}(n)$, for $y_{\mathbf{y}_1}$ this takes $\mathcal{O}(2^{n/2 - \log n})$. To avoid this, we use the techniques of the delegation step of **Phase 1**, to establish the correctness and well-formedness of $f_{\mathbf{y}_1}$ (step 3).

The following is proven in Appendix A where we also show how it can be made zero-knowledge adapting the techniques of [10].

Theorem 1 *Let $f, g : \mathbb{F}^n \rightarrow \mathbb{F}$ be multi-linear polynomials. Our construction is a space-efficient argument of knowledge for the sumcheck instance $K = \sum_{x \in \{0, 1\}^n} f(x)g(x)$. The prover requires space of $\mathcal{O}(\sqrt{2^n})$ field elements, and has proving complexity of $\mathcal{O}(2^n \log n)$. Proof size and verification complexity are $\mathcal{O}(n)$.*

3.2 Our Space-Efficient Argument of Knowledge

In this section, we construct our space-efficient argument of knowledge for data-parallel arithmetic circuits. For our purposes, we consider a d -layered arithmetic circuit \mathcal{C} to be data-parallel if, for every layer $i \in [d]$, its corresponding circuit denoted with \mathcal{C}_i , consists of multiple copies of sub-circuits running on different inputs [67, 68, 82, 28, 8, 69]. Similar to prior works, we assume each layer consists of multiple copies of a single sub-circuit, denoted with \mathcal{C}'_i , but we can directly extend this for different sub-circuits [82].

An arithmetic circuit C can be evaluated naturally in optimal time $\Theta(|C|)$. For arbitrary circuits, in order to achieve this optimal evaluation time one may need $O(|C|)$ space to store the circuit topology and partial gate evaluations. Indeed, prior works for space-efficient arguments that support arbitrary circuits [26] require this much storage. In contrast, by focusing on data-parallel arithmetic circuits, we achieve prover space $\mathcal{O}(sp^S(\mathbf{x}) + \sum_{i \in [d]} |\mathcal{C}'_i|) + \sqrt{|C|}$ where $sp^S(\mathbf{x})$ is the space required to instantiate the streaming oracle $S(\mathbf{x})$ for the input \mathbf{x} .² That is, we achieve the first space-efficient SNARK for arithmetic circuits with prover space asymptotically smaller than the optimal circuit evaluation time—albeit for data-parallel circuits.

To build our space-efficient SNARK, we first construct a space-efficient variant of the GKR protocol that is given a “short” description of \mathcal{C} (e.g., the description of the sub-circuits and the number of copies per layer), streaming access to \mathbf{x} , and proves the correct computation of the circuit. As a first step, we construct a space-efficient protocol for the correct computation of a single layer of the circuit (see Section 3.2.1), which we later use to realize a space-efficient variant of the GKR protocol (Section 3.2.2). Finally, in Section 3.2.3 we compile the latter into an argument of knowledge.

3.2.1 Proving the correct computation of a single layer.

Recall from Section 2 that the GKR protocol, starting from the output, successively proves the correct computation of every layer until reaching the input. Hence, to build a space-efficient variant of GKR, we first need to construct a space-efficient protocol for proving the correct computation of a single layer. More precisely, assuming that S_i (and S'_i resp.) is the number of input gates of \mathcal{C}_i (and \mathcal{C}'_i resp.), $s_i = \log S_i$, $s'_i = \log S'_i$, \mathbf{V}_i the input of \mathcal{C}_i (with $\mathbf{x} = \mathbf{V}_d$ corresponding to the input of the \mathcal{C}) and f_i its multi-linear extension, we want to prove the following sumcheck instance:

$$f_{i-1}(r) = \sum_{(x,y) \in \{0,1\}^{2s_i}} \left(f_{add}^i(r, x, y)(f_i(x) + f_i(y)) + f_{mul}^i(r, x, y)f_i(x)f_i(y) \right)$$

Where $r \in \mathbb{F}^{s_{i-1}}$ is a randomly selected point and $i \in [d]$ a layer of \mathcal{C} . At first glance, this instance seems incompatible with the protocol we introduced in the previous section, as it contains multiplications of more than two polynomials. Fortunately, by relying on the techniques introduced in [10], we can divide our protocol into two phases, where in every phase, we generate a proof for a sumcheck of the form $\sum_{x \in \{0,1\}^n} f(x)g(x)$. Then, we invoke our space-efficient protocol to prove that instance. Due to space limitations, we will only present how to deal with the multiplication part of the sumcheck, e.g., generating a proof for $K = \sum_{(x,y) \in \{0,1\}^{2n}} f_{mul}^i(r, x, y)f_i(x)f_i(y)$, but we emphasize that we can trivially handle additions in a likely manner.

Phase 1. In this phase, we need to generate a proof for $K = \sum_{x \in \{0,1\}^{s_i}} f_i(x)h(x)$, where $h(x) = \sum_{y \in \{0,1\}^{s_i}} f_{mul}^i(r, x, y)f_i(y)$ (see Section 3.3 of [10]). Assuming that h is the multi-linear extension of the vector \mathbf{A}_h , we can prove the latter instance by directly invoking our space efficient sumcheck

²Note that $sp^S(\mathbf{x})$ is always upper-bounded by $|\mathbf{x}|$. Looking ahead, we will see cases in which $sp^S(\mathbf{x})$ is sub-linear to $|\mathbf{x}|$.

protocol with input the streaming oracles $S(\mathbf{V}_i), S(\mathbf{A}_h)$. What remains is to show how to instantiate these oracles in practice.

Instantiating $S(\mathbf{V}_i)$. To instantiate streaming access to \mathbf{V}_i , we need to construct a routine which, upon the j -th invocation, outputs the vector $\mathbf{V}_i[(j-1)S'_i : jS'_i]$. To achieve that, we will invoke $S(\mathbf{V}_{i+1})$ to receive the next sub-array of \mathbf{V}_{i+1} , $\tilde{\mathbf{V}}_{i+1} \in \mathbb{F}^{S'_{i+1}}$. Then evaluate the sub-circuit $\mathcal{C}'_{i+1}(\tilde{\mathbf{V}}_{i+1})$, storing the output values into the buffer. If the output size is less than S'_i , we repeat the same process until copying S'_i elements. Note that the base step of this recursion is $i = d$, where in the j -th invocation we output $\mathbf{V}_d[jS'_d : (j+1)jS'_d]$.

Instantiating $S(\mathbf{A}_h)$. Upon the j -th invocation, $S(\mathbf{A}_h)$ outputs the evaluations of $h(\mathbf{j}, x_2)$ in the hypercube with $\mathbf{j} \in \{0, 1\}^{s_i - s'_i}$. Because our circuit is data-parallel, we can re-write h as:

$$h(\mathbf{j}, x_2) = \beta(r_1, \mathbf{j}) \cdot \sum_{z_2, y_2 \in \{0, 1\}^{s'_{i-1} + s'_i}} \beta(r_2, z_2) f_{mul}^i(z_2, x_2, y_2) f_i(\mathbf{j}, y_2)$$

where $r = (r_1, r_2) \in \mathbb{F}^{s_{i-1} - s'_{i-1}} \times \mathbb{F}^{s'_{i-1}}$ and f_{mul}^i the multi-linear extension of the multiplication wiring predicate of \mathcal{C}'_i . We can efficiently evaluate the latter using $S(\mathbf{V}_i)$ and computing “on the fly” the necessary evaluations of $\beta(\cdot)$.

Phase 2. After the first phase, we have evaluations of f_i and h at a random point $r' \in \mathbb{F}^{s_i}$. In the second phase, we must generate a proof for $h(r') = \sum_{y \in \{0, 1\}^{s_i}} g(y) f_i(y)$, where $g(y) = f_{mul}^i(r, r', y)$, is the multi-linear extension of array \mathbf{A}_g . As before, we generate this proof using our space-efficient protocol given oracles $S(\mathbf{A}_g), S(\mathbf{V}_i)$. Since we discussed how to construct $S(\mathbf{V}_i)$, we will solely focus on instantiating streaming access to \mathbf{A}_g .

Instantiating $S(\mathbf{A}_g)$. To establish streaming access to \mathbf{A}_g , we need to construct a routine which on the j -th invocation outputs $\mathbf{A}_g[(j-1)2^{s'_i} : j2^{s'_i}]$. This translates to efficiently computing the polynomial $g(\mathbf{j}, y_2)$ in all $y_2 \in \{0, 1\}^{s'_i}$. Due to the parallel nature of our circuit we can re-write g as:

$$g(\mathbf{j}, y_2) = \beta(r_1, \mathbf{j}) \beta(r'_1, \mathbf{j}) \sum_{z_2, x_2 \in \{0, 1\}^{s'_{i-1} + s'_i}} \beta(r_2, z_2) \beta(r'_2, x_2) f_{mul}^i(z_2, x_2, y_2)$$

Note that we can efficiently compute all evaluations of this polynomial “on the fly”.

Regarding space efficiency, observe that to instantiate $S(\mathbf{V}_i)$, we need space of $sp^S(\mathbf{x})$ field elements (because for the baseline step of the recursion, we need to have access to \mathbf{V}_d) and an additional space of $\sum_{j \in [i, d-1]} (|\mathcal{C}'_{j+1}| + |\mathcal{C}'_j|)$ (to compute the output of every recursive invocation). We follow a similar argument for $S(\mathbf{A}_h), S(\mathbf{A}_g)$. Moreover, from Theorem 1, we know that the space required to generate a proof for the sumcheck protocol is $\mathcal{O}(\sqrt{S_i})$, leading to an overall space complexity of $\mathcal{O}(sp^S(\mathbf{x}) + \sqrt{S_i} + \sum_{j \in [i, d]} |\mathcal{C}'_j|)$. As for the proving complexity, we invoke our sumcheck protocol twice and for each invocation, we need to scan $S(\mathbf{V}_i), S(\mathbf{A}_g), S(\mathbf{A}_h)$ a constant number of times. A single scan over $S(\mathbf{V}_i)$ (and consequently $S(\mathbf{A}_h)$) requires $\sum_{j \in [d, i]} S_j$ field operations as we need to evaluate \mathcal{C} until layer i , while for $S(\mathbf{A}_g)$ we only need $\mathcal{O}(S_i)$. Thus the total proving complexity is $\mathcal{O}(S_i \log \log S_i + \sum_{j \in [d, i]} S_j)$.

3.2.2 Proving the correct computation of \mathcal{C} .

We can now directly instantiate a space-efficient variant of GKR replacing its per-layer sub-protocol (see Section 3.3 of [10]) with the one described above. **Protocol 2** gives a detailed description of our space-efficient variant of the GKR protocol. Observe that in the main previous section, we described how to prove a sumcheck instance only for step 3. However, this instance accounts only for the output layer. In practice, because we need to reduce two evaluation points into one, for all other layers we end up with a sumcheck instance as presented in step 6.b.

Protocol 2 : Space-Efficient Variant of GKR. Let \mathcal{C} be a d -layered data-parallel arithmetic circuit. We denote with $\mathbf{V}_d \in \mathbb{F}^{S_d}$ the input values of \mathcal{C} , $\mathbf{V}_0 \in \mathbb{F}^{S_0}$ its output values and $\mathbf{V}_i \in \mathbb{F}^{S_i}$ the output values of the i -th layer of \mathcal{C} . Furthermore, we denote with $f_i : \mathbb{F}^{S_i} \rightarrow \mathbb{F}$ the multi-linear extension of \mathbf{V}_i . Assume a prover \mathcal{P} having streaming access to \mathbf{V}_d and verifier \mathcal{V} having oracle access to f_d, f_0 . \mathcal{P} proves that $\mathcal{C}(\mathbf{V}_d) = \mathbf{V}_0$ by interacting with \mathcal{V} over the following protocol:

1. \mathcal{V} : Select a random point $\mathbf{r}^{(0)} \in \mathbb{F}^{S_0}$ and send it to \mathcal{P} .
2. \mathcal{P} : Evaluates $f_0(\mathbf{r}^{(0)})$ using access to the streaming oracle $S(\mathbf{V}_0)$ and sends $f_0(\mathbf{r}^{(0)})$ to \mathcal{V} .
3. \mathcal{P} - \mathcal{V} : Interact following the space-efficient protocol (as described in Section 3.2.1) for proving the correctness of the output layer:

$$f_d(\mathbf{r}^{(0)}) = \sum_{x,y \in \{0,1\}^{2s_1}} f_{add}^i(\mathbf{r}^{(0)}, x, y)(f_1(x) + f_1(y)) + f_{mul}^i(\mathbf{r}^{(0)}, x, y)(f_1(x)f_1(y))$$

4. \mathcal{P} : At the end of the protocol, send $f_1(\mathbf{r}_1^{(1)}), f_1(\mathbf{r}_2^{(1)})$ to \mathcal{V} .
5. \mathcal{V} : Evaluate $u_{add} = f_{add}^i(\mathbf{r}^{(0)}, \mathbf{r}_1^{(1)}, \mathbf{r}_2^{(1)})$, $u_{mul} = f_{mul}^i(\mathbf{r}^{(0)}, \mathbf{r}_1^{(1)}, \mathbf{r}_2^{(1)})$, and check if the last round of sumcheck equals to $u_{add} \cdot (f_1(\mathbf{r}_1^{(1)}) + f_1(\mathbf{r}_2^{(1)})) + u_{mul} \cdot f_1(\mathbf{r}_1^{(1)}) \cdot f_1(\mathbf{r}_2^{(1)})$.
6. For $i = 2, \dots, d$ do:

- (a) \mathcal{V} : Randomly selects $a, b \in \mathbb{F}$ and sends them to \mathcal{P} .
- (b) \mathcal{P} - \mathcal{V} : Interact following the space-efficient protocol (as described in Section 3.2.1) for proving the correctness of the $(i-1)$ -th layer:

$$af_{i-1}(\mathbf{r}_1^{(i-1)}) + bf_{i-1}(\mathbf{r}_2^{(i-1)}) = \sum_{x \in \{0,1\}^{s_i}} F_{add}(x, y)(f_i(x) + f_i(y)) + F_{mul}(x, y)(f_i(x)f_i(y))$$

Where $F_{add}(x, y) = a \cdot f_{add_i}(\mathbf{r}_1^{(i-1)}, x, y) + b \cdot f_{add_i}(\mathbf{r}_2^{(i-1)}, x, y)$, $F_{mul}(x, y) = a \cdot f_{mul_i}(\mathbf{r}_1^{(i-1)}, x, y) + b \cdot f_{mul_i}(\mathbf{r}_2^{(i-1)}, x, y)$

- (c) \mathcal{P} : At the end of the protocol, send $f_i(\mathbf{r}_1^{(i)}), f_i(\mathbf{r}_2^{(i)})$ to the verifier.
- (d) \mathcal{V} : Compute the evaluations of the wiring predicates and validate the last round of the sumcheck as in step (5).

7. \mathcal{V} : Picks $a_1, a_2 \in \mathbb{F}$ at random and sends it to the \mathcal{P} .
8. \mathcal{P} - \mathcal{V} : Interact following the space-efficient sumcheck protocol (**Protocol 1** of Section 3.1) Protocol 1 for the instance:

$$a_1 f_d(\mathbf{r}_1^{(d)}) + a_2 f_d(\mathbf{r}_2^{(d)}) = \sum_{x \in \{0,1\}^{s_d}} (a_1 \beta(x, \mathbf{r}_1^{(d)}) + a_2 \beta(x, \mathbf{r}_2^{(d)})) f_d(x)$$

Using as input the streaming oracles $S(\mathbf{V}_d)$ and $S(\mathbf{G})$ where \mathbf{G} stores the evaluations of the polynomial $a_1 \beta(x, \mathbf{r}_1^{(d)}) + a_2 \beta(x, \mathbf{r}_2^{(d)})$ at all $x \in \{0,1\}^{s_d}$. Finally, send $f_d(\mathbf{r}_d)$ to \mathcal{V} .

9. \mathcal{V} : Validates the last sumcheck round by evaluating $\beta(\mathbf{r}_d, \mathbf{r}_1^{(d)}), \beta(\mathbf{r}_d, \mathbf{r}_2^{(d)})$. Finally, it queries the oracles of the polynomials f_0, f_d at $\mathbf{r}^{(0)}$ and \mathbf{r}_d respectively to validate the correctness of $f_0(\mathbf{r}^{(0)})$ and $f_d(\mathbf{r}_d)$.

Although this is almost identical with the one of step 3, we need to perform some additional work when instantiating streaming access to the hypercube evaluations of $h(x)$ and $g(x)$ (denoted

with \mathbf{A}_h and \mathbf{A}_g). Starting from $h(x)$ observe that:

$$\begin{aligned} h(x) &= \sum_{y \in \{0,1\}^{s_i}} (af_{mul}(\mathbf{r}_1^{(i)}, x, y) + bf_{mul}(\mathbf{r}_2^{(i)}, x, y))f_i(y) = \\ a \sum_{y \in \{0,1\}^{s_i}} f_{mul}(\mathbf{r}_1^{(i)}, x, y)f_i(y) &+ b \sum_{y \in \{0,1\}^{s_i}} f_{mul}(\mathbf{r}_2^{(i)}, x, y)f_i(y) = \\ &ah_1(x) + bh_2(x) \end{aligned}$$

So, to instantiate streaming access to \mathbf{A}_h , we create a routine which on the j -th invocation calls the streaming oracles of \mathbf{A}_{h_1} and \mathbf{A}_{h_2} , as we described in section 3.2.1, and uses a, b to aggregate the results. Similarly, we generate the streaming access to \mathbf{A}_g .

Finally, note that at the end of the step 6 of **Protocol 2**, the verifier ends up with two evaluations of the f_d , at $\mathbf{r}_1^{(d)}, \mathbf{r}_2^{(d)}$. We can reduce these claims into one by applying an additional space-efficient sumcheck (step 8). For that sumcheck we need streaming access to the vector \mathbf{B} that contains all evaluations of $(a_1\beta(x, \mathbf{r}_1^{(d)}) + a_2\beta(x, \mathbf{r}_2^{(d)}))$ at $x \in \{0, 1\}^{s_d}$. To efficiently instantiate such access, we can precompute the tables $\mathbf{B}_{1,1} = (\beta(1, \mathbf{r}_{1,1}^{(d)}), \dots, \beta(\sqrt{2^{s_d/2}}, \mathbf{r}_{1,1}^{(d)}))$, $\mathbf{B}_{1,2} = (\beta(1, \mathbf{r}_{1,2}^{(d)}), \dots, \beta(\sqrt{2^{s_d/2}}, \mathbf{r}_{1,2}^{(d)}))$ (and $\mathbf{B}_{2,1}, \mathbf{B}_{2,2}$ resp.), where $\mathbf{r}_{1,2}^{(d)} = (\mathbf{r}_{1,1}^{(d)}, \mathbf{r}_{1,2}^{(d)}) \in \mathbb{F}^{s_d/2} \times \mathbb{F}^{s_d/2}$ and compute each evaluation on the fly using only two multiplications.

The overall space complexity is the maximum space required for a single layer, i.e., $\mathcal{O}(sp^S(\mathbf{x}) + \sum_{i \in [d]} |\mathcal{C}'_i| + \sqrt{S})$ (where $S = \max(\{S\}_{i \in [d]})$), which is the space needed to evaluate \mathcal{C} plus an additional \sqrt{S} . Also, the proving complexity is $\sum_{i \in [d]} (S_i \log \log S_i + \sum_{j \in [i]} S_j) \approx \mathcal{O}(|\mathcal{C}| \log \log |\mathcal{C}| + d|\mathcal{C}|)$. Unfortunately, this grows with the circuit depth, which could lead to increased proving times [8, 28]. To circumvent that, we use the following lemma which we prove in Appendix B:

Lemma 1 *We can convert a d -layered data-parallel arithmetic circuit \mathcal{C} into a \tilde{d} -layered one $\tilde{\mathcal{C}}$ of size $\mathcal{O}(|\mathcal{C}|)$ and $\tilde{d} < \log \log |\mathcal{C}|$, consisting of multiple copies of d distinct sub-circuits such that $|\tilde{\mathcal{C}}'_i| \leq 2|\mathcal{C}'_i|, i \in [d]$. Moreover, we can perform a pass over $S(\tilde{\mathbf{x}})$ in time $|\mathcal{C}|$ and space $sp^S(\tilde{\mathbf{x}}) = sp^S(\mathbf{x}) + \sum_{j \in [i,d]} |\mathcal{C}'_j|$. Finally, if \mathcal{C} is log-space uniform then $\tilde{\mathcal{C}}$ is also log-space uniform.*

This allows us to generate a proof for the correct computation of \mathcal{C} by invoking the space-efficient GKR protocol over the “squashed” circuit $\tilde{\mathcal{C}}$ reducing the proving complexity. Indeed, since $\tilde{\mathcal{C}}$ is data-parallel, we can evaluate it in space $sp^S(\tilde{\mathbf{x}}) + \sum_{i \in [d]} |\tilde{\mathcal{C}}'_i|$, and thus we can prove its correct computation with $\mathcal{O}(sp^S(\tilde{\mathbf{x}}) + \sum_{i \in [d]} |\tilde{\mathcal{C}}'_i| + \sqrt{|\mathcal{C}|})$ space complexity, and $\mathcal{O}(|\mathcal{C}| \log \log |\mathcal{C}|)$ proving complexity.

3.2.3 Our Space-Efficient Argument of Knowledge

Next, we present our space-efficient argument of knowledge. Given a data-parallel circuit \mathcal{C} , instance \mathbf{x} and streaming access to witness \mathbf{w} , we want to generate a proof for the relation $\mathcal{R} = \{(\mathcal{C}, \mathbf{x}; \mathbf{w}) : \mathcal{C}(\mathbf{x}, \mathbf{w}) = 1\}$. Our protocol, presented in **Construction 1**, follows the same strategy as prior (space-inefficient) GKR-based zkSNARKs [83, 10, 28, 84]. First, we compute the polynomial commitment of the multi-linear extension of \mathbf{w} , $f_{\mathbf{w}}$, using our space-efficient PC scheme with input the streaming oracle $S(\mathbf{w})$ (step 1). Then, in step 2 we prove the correct computation of \mathcal{C} by invoking our space-efficient GKR protocol (as presented in **Protocol 2**) using streaming access to its input $\mathbf{V}_d = (\mathbf{x}, \mathbf{w})$. Finally, step 3 we must prove the correctness of claimed evaluations y_x, y_w of $f_{\mathbf{x}}, f_{\mathbf{w}}$ at a random point. The verifier validates y_x locally, and y_w using a PC evaluation proof generated by the prover.

Construction 1 : Sparrow. Let λ be a security parameter, \mathbb{F} be a finite field and N be an upper bound of circuit input size. The following construction presents the interactive version of our space-efficient argument of knowledge.

- $Gen(1^\lambda, N)$: Invoke $(\mathbf{pk}, \mathbf{vk}) \leftarrow PC.Gen(1^\lambda, \log n, 1)$ and send \mathbf{pk} to \mathcal{P} and \mathbf{vk} to \mathcal{V} .
- $\langle \mathcal{P}(\mathbf{pk}, \mathbf{w}), \mathcal{V}(\mathbf{vk}) \rangle(\mathbf{x})$: Let \mathcal{C} be a d -layered data-parallel arithmetic circuit, \mathbf{x} be a statement and \mathbf{w} be a witness such that $\mathcal{C}(\mathbf{x}, \mathbf{w}) = 1$ and $|\langle \mathbf{x}, \mathbf{w} \rangle| \leq N$. Without loss of generality, we assume that $|\mathbf{w}|/|\mathbf{x}| = 2^m$.
 1. \mathcal{P} : Computes and sends to \mathcal{V} , $C_w \leftarrow PC.Commit(\mathbf{pp}, S(\mathbf{V}_d))$, using the space-efficient PC scheme, where $\mathbf{V}_d = (\mathbf{x}, \mathbf{w})$.
 2. \mathcal{P} interacts with \mathcal{V} following the **Space-Efficient Variant of the GKR** protocol to prove that $\mathcal{C}(\mathbf{x}, \mathbf{w}) = 1$. At the end of the GKR protocol, both parties end up with a claimed evaluation y of f_d over \mathbf{r} .
 3. \mathcal{P} Invokes $\pi, y_w \leftarrow PC.Eval(\mathbf{pp}, S(\mathbf{x}), \mathbf{r}_w)$, where $\mathbf{r}_w = (r_1, \dots, r_{\log|\mathbf{w}|}, \mathbf{0}^{\log(|\mathbf{x}|+|\mathbf{w}|)-\log|\mathbf{w}|})$ and sends π, y_w to \mathcal{V} .
 4. \mathcal{V} : Evaluates $y_x = f_x(\mathbf{r}_x)$, on $\mathbf{r}_x = (r_1, \dots, r_{\log|\mathbf{x}|})$, checks if $PC.Verify(\mathbf{vk}, \pi, C_w, \mathbf{r}_w) = 1$ and validates y using y_w, y_x .

What remains, is to find a PC scheme that satisfies our space requirements. Note that we cannot use multi-linear KZG, as GEMINI does, since its public parameters scale linearly to the witness size (or even to \mathcal{C} if depth reduction is used). To overcome that issue, we need a scheme with public parameters sub-linear to the polynomial size. Fortunately, such schemes already exist [85, 35, 86]. For SPARROW, we will use Kopsis [85], a scheme based on inner-pairing products [86] and KZG, that has public parameters of size $\mathcal{O}(\sqrt{N})$ group elements, commitment complexity of \sqrt{N} MSM of size \sqrt{N} and \sqrt{N} pairings, evaluation complexity of $\mathcal{O}(N)$ field operations and $\mathcal{O}(\sqrt{N})$ pairing operations, and $\mathcal{O}(\log N)$ proof size and verification complexity. Furthermore, we can instantiate a space-efficient variant for the commit and evaluation algorithms straightforwardly, using $\mathcal{O}(\sqrt{N})$ buffer space and maintaining the same performance characteristics.

In particular, assuming that $f : \mathbb{F}^{\log N} \rightarrow \mathbb{F}$ is the multi-linear extension of $\mathbf{A}_f \in \mathbb{F}^N$ for which we have streaming access, we can instantiate a space efficient variant of the commit and evaluation algorithms of Kopsis in the following way. To generate the commitment C_f , we first scan $S(\mathbf{A}_f)$, reading the next \sqrt{N} elements, use them to compute the polynomial $f_i : \mathbb{F}^{\log N/2} \rightarrow \mathbb{F}$ and its KZG commitment C_i which we store in a buffer. After reading all elements, we compute $C_f = \prod_{i \in [\sqrt{N}]} e(C_i, v_i)$. To generate an evaluation proof at the point $r = (r_1, r_2) \in \mathbb{F}^{\log N/2} \times \mathbb{F}^{\log N/2}$, we use the inner-pairing product introduced in [86] to prove that $C^* = \prod_{i \in [\sqrt{N}]} C_i^{\beta(i, r_1)}$. Finally, we scan \mathbf{A}_f one more time, to evaluate the aggregated polynomial $f^*(x) = \sum_{i \in \{0,1\}^{\log N/2}} \beta(i, r_1) f_i(x)$ and generate an evaluation proof for f^* at r_2 , using the KZG evaluation algorithm.

Note that we can make the argument zero-knowledge with the standard technique of [10], combined with the zero-knowledge version of our sumcheck 3.1, and Kopsis [85]. Also observe that by applying the Fiat-Shamir heuristic, our protocol becomes non-interactive. Finally, we can state the following result which we prove in Appendix B.

Theorem 2 For a d -layered data-parallel arithmetic circuit \mathcal{C} , SPARROW is a zero-knowledge argument of knowledge for relation $\mathcal{R} = \{(\mathcal{C}, \mathbf{x}, \mathbf{w}) : \mathcal{C}(\mathbf{x}, \mathbf{w}) = 1\}$, with proving complexity of $\mathcal{O}(|\mathcal{C}| \log \log |\mathcal{C}|)$ field operations and MSM of size $\mathcal{O}(|\mathcal{C}|)$, $\mathcal{O}(|\mathbf{x}| + sp^S(\mathbf{w}) + \sum_{i \in [d]} |\mathcal{C}'_i| + \sqrt{|\mathcal{C}|})$ space, and $\mathcal{O}(\log(|\mathcal{C}|))$ proof size. For a log-space uniform \mathcal{C} , the verification time is $\mathcal{O}(|\mathbf{x}| + \log |\mathcal{C}|)$.

4 Zero-Knowledge Proofs of Forest Training and Predictions

In this section, we introduce our scheme for Zero-Knowledge proofs of forest training and predictions (*zkFTP*), which enables a prover to commit to a dataset D and a forest \mathcal{F} and prove in zero-knowledge that \mathcal{F} has been trained correctly on D . Later, given test point \mathbf{x} it can be proved in zero-knowledge that y is the corresponding prediction with respect to \mathcal{F} . We define *zkFTP* as a tuple of the probabilistic algorithms (*KeyGen*, *ComData*, *ComForest*, *ProveTrain*, *VerifyTrain*, *ProvePred*, *VerifyPred*). At a high level, *KeyGen* takes as input the dimensions of the dataset, maximum height h , and number of trees K , and generates proving and verification public parameters. *ComData* takes as input $D \in [B]^{n \times d}$ and computes the commitment C_D . Likewise, we define *ComForest*. Given D and \mathcal{F} , *ProveTrain* generates a proof that $\mathcal{F} = \text{TRAIN}(D)$. *VerifyTrain* uses this proof along with $C_D, C_{\mathcal{F}}$ to validate its correctness. Similarly, *ProveForest* takes as input \mathcal{F} , a test point \mathbf{x} and returns y along with a prediction proof. Finally, *VerifyPred* uses the latter proof along with $C_{\mathcal{F}}$ to check whether $y = \text{PREDICT}(\mathcal{F}, \mathbf{x})$. *zkFTP* must satisfy *forest and data extractability*, meaning that if a prover generates an accepting proof then it must know a forest \mathcal{F} trained on data D and *seed*, and both \mathcal{F}, D are the pre-images of $C_{\mathcal{F}}$ and C_D , respectively. It also achieves *training zero-knowledge*, i.e., proof π_T reveals no additional information about \mathcal{F} and D other than the dimensions of D , maximum height, total number of nodes and trees in \mathcal{F} . It achieves similar properties for prediction, for which we follow the formulation of [40]. Formally, our *zkFTP* is described by the following algorithms:

- $\mathbf{pk}, \mathbf{vk} \leftarrow \text{KeyGen}(1^\lambda, n, d, h, K)$: Given a security parameter λ , maximum number of points n , number of features d , maximum tree height h and number of trees K in forest, generate public parameters \mathbf{pk} (for the prover) and \mathbf{vk} (for the verifier).
- $C_D \leftarrow \text{ComData}(\mathbf{pk}, D, r_D)$: Given a dataset $D \in [B]^{n \times d}$ and randomness r_D , return a dataset commitment C_D .
- $C_{\mathcal{F}} \leftarrow \text{ComForest}(\mathbf{pk}, \mathcal{F}, r_F)$: Given forest \mathcal{F} of K trees $\{\mathcal{T}_1, \dots, \mathcal{T}_K\}$ and randomness r_F , return dataset commitment $C_{\mathcal{F}}$.
- $\pi_T \leftarrow \text{ProveTrain}(\mathbf{pk}, D, \mathcal{F}, r_D, r_F, \text{seed})$: Return a proof π_T showing that $\mathcal{F} \leftarrow \text{TRAIN}(D, \text{seed})$, where *seed* bootstraps the training randomness.
- $0, 1 \leftarrow \text{VerifyTrain}(\mathbf{vk}, C_D, C_{\mathcal{F}}, \pi_T, \text{seed})$: Return 1 if π_T is valid proof of forest training with respect to the pre-images of $C_D, C_{\mathcal{F}}$ and *seed*.
- $y, \pi_P \leftarrow \text{ProvePred}(\mathbf{pk}, \mathcal{F}, r_F, \mathbf{x})$: Return a proof π_P showing that $y = \text{PREDICT}(\mathcal{F}, \mathbf{x})$.
- $0, 1 \leftarrow \text{VerifyPred}(\mathbf{vk}, C_{\mathcal{F}}, \pi_P, y, \mathbf{x})$: Validate the prediction y on \mathbf{x} using π_P and $C_{\mathcal{F}}$.

Regarding the training phase, our scheme must satisfy *training completeness*, meaning that the prover always generates a valid proof for a correctly trained forest \mathcal{F} . In addition, it must satisfy *forest and data extractability* in the sense that whenever a prover generates an accepting proof of forest training then it has to know the pre-images of $C_{\mathcal{F}}$ and C_D , \mathcal{F} and D respectively, such that $\mathcal{F} = \text{TRAIN}(D, \text{seed})$ with overwhelming probability. Finally, our scheme must satisfy *training zero-knowledge*, meaning that the proof π_T reveals nothing about the forest and the dataset other than the information the verifier already knows (namely, the dataset size, number of features, number of trees, and maximum depth of the trees). More formally:

- **Training Completeness.** For any \mathbf{pk}, \mathbf{vk} generated by $KeyGen$, dataset $D \in [B]^{n \times d}$, $seed$, $C_D \leftarrow ComData(D, r_D)$ it holds that:

$$Pr \left(\begin{array}{l} \pi_T, C_{\mathcal{F}} \leftarrow ProveTrain(\mathbf{pk}, D, \mathcal{F}, r_D, r_F, K, seed) \wedge \\ \mathcal{F} = \text{TRAIN}(D, seed) : \\ 1 \leftarrow VerifyTrain(\mathbf{vk}, C_D, C_{\mathcal{F}}, \pi_T) \end{array} \right) = 1$$

- **Forest and Data Extractability.** For any PPT adversary \mathcal{A} , there exist a PPT extractor \mathcal{E} such that:

$$Pr \left(\begin{array}{l} C_D, C_{\mathcal{F}}, \pi_T, seed \leftarrow \mathcal{A}(\mathbf{pk}, \mathbf{vk}) \wedge \\ \mathcal{F}, D, r_D, r_F \leftarrow \mathcal{E}(\mathbf{pk}, \mathbf{vk}) \wedge \\ 1 \leftarrow VerifyTrain(\mathbf{vk}, C_D, C_{\mathcal{F}}, \pi_T, seed) \wedge \\ (\mathcal{F} \neq \text{TRAIN}(D, seed)) \vee \\ (C_D \neq ComData(D, r_D) \neq ComForest(\mathcal{F}, r_F)) \end{array} \right) \leq \text{negl}(\lambda)$$

- **Training Zero-Knowledge.** For any number of instances n , features d , trees K with maximum depth h , \mathbf{pk}, \mathbf{vk} produced by $KeyGen$, and adversary \mathcal{A} , there exists a simulator \mathcal{S} such that for the experiments $Real_{\mathcal{A}, D, \mathcal{F}}(\mathbf{pk}, \mathbf{vk})$, $Ideal_{\mathcal{A}, n, K, h, seed}(\mathbf{pk}, \mathbf{vk})$ (as defined bellow) we have:

$$P(Real_{\mathcal{A}, D, \mathcal{F}}(\mathbf{pk}, \mathbf{vk}) = 1) \approx P(Ideal_{\mathcal{A}, n, K, h, seed}(\mathbf{pk}, \mathbf{vk}) = 1)$$

$Real_{\mathcal{A}, D, \mathcal{F}}(\mathbf{pk}, \mathbf{vk})$

1. $seed \leftarrow \mathcal{A}(\mathbf{pk}, \mathbf{vk})$
2. $C_D \leftarrow ComData(\mathbf{pk}, D, r_D)$
3. $C_{\mathcal{F}} \leftarrow ComForest(\mathbf{pk}, \mathcal{F}, r_F)$
4. $\pi_T \leftarrow ProveTrain(\mathbf{pk}, D, \mathcal{F}, r_D, r_F, K, seed)$
5. $b \leftarrow \mathcal{A}(\mathbf{vk}, C_D, C_{\mathcal{F}}, \pi_T)$
6. Return b

$Ideal_{\mathcal{A}, n, d, K, h}(\mathbf{pk}, \mathbf{vk}, \text{trap})$

1. $seed, n, d, h, K \leftarrow \mathcal{A}(\mathbf{pk}, \mathbf{vk})$
2. $C_D \leftarrow S_1(\mathbf{pk}, \text{trap}, n, d)$
3. $C_{\mathcal{F}} \leftarrow S_2(\mathbf{pk}, \text{trap}, K, h, seed)$
4. $\pi_T \leftarrow S_3(\mathbf{pk}, \text{trap}, C_D, C_{\mathcal{F}}, seed)$
5. $b \leftarrow \mathcal{A}(\mathbf{vk}, C_D, C_{\mathcal{F}}, \pi_T)$
6. Return b

As for the prediction phase, we want $zkFTP$ to satisfy *prediction completeness* in which the prover will always generate an accepting proof for a correct prediction. Moreover, we require *forest extractability*, meaning that whenever a prover generates an accepting prediction proof, it must know a forest \mathcal{F} such that $y = \text{PREDICT}(\mathcal{F}, \mathbf{x})$ with overwhelming probability. Lastly, $zkFTP$ needs to satisfy *prediction zero-knowledge* in the sense that π_P reveals nothing about \mathcal{F} other than what the verifier already knows, which is the number of trees K , the maximum height of trees h and the prediction of \mathcal{F} at \mathbf{x} . Specifically, we have:

- **Prediction Completeness.** For any \mathbf{pk}, \mathbf{vk} , forest \mathcal{F} , $C_{\mathcal{F}} \leftarrow ComForest(D, r_D)$ and test point \mathbf{x} it holds that:

$$Pr \left(\begin{array}{l} \pi_P, y \leftarrow ProvePred(\mathbf{pk}, \mathcal{F}, r_F, \mathbf{x}) \wedge \\ \mathcal{F} = \text{PREDICT}(\mathcal{F}, \mathbf{x}) \wedge \\ 1 \leftarrow VerifyPred(\mathbf{vk}, C_{\mathcal{F}}, \pi_P, y, \mathbf{x}) \end{array} \right) = 1$$

- **Forest Extractability.** For any adversary \mathcal{A} , there exist an extractor \mathcal{E} such that:

$$Pr \left(\begin{array}{l} C_{\mathcal{F}}, \pi_P, \mathbf{x}, y \leftarrow \mathcal{A}(\mathbf{pk}, \mathbf{vk}) \wedge \\ \mathcal{F}, r_F \leftarrow \mathcal{E}(\mathbf{pk}, \mathbf{vk}, C_D, C_{\mathcal{F}}, \pi_T) \wedge \\ 1 \leftarrow \text{VerifyPred}(\mathbf{vk}, C_{\mathcal{F}}, \pi_P, y, \mathbf{x}) \wedge \\ (y \neq \text{PREDICT}(\mathcal{F}, \mathbf{x}) \vee C_{\mathcal{F}} \neq \text{ComData}(\mathcal{F}, r_F)) \end{array} \right) \leq \text{negl}(\lambda)$$

- **Prediction Zero-Knowledge.** For any number of trees K with maximum depth h , \mathbf{pk}, \mathbf{vk} , adversary \mathcal{A} , there exists a simulator \mathcal{S} such that for the experiments $\text{Real}_{\mathcal{A}, \mathcal{F}}(\mathbf{pk}, \mathbf{vk})$, $\text{Ideal}_{\mathcal{A}, K, h}(\mathbf{pk}, \mathbf{vk})$ (as defined bellow) we have:

$$Pr(\text{Real}_{\mathcal{A}, \mathcal{F}}(\mathbf{pk}, \mathbf{vk}) = 1) \approx Pr(\text{Ideal}_{\mathcal{A}, K, h}(\mathbf{pk}, \mathbf{vk}) = 1)$$

<u>$\text{Real}_{\mathcal{A}, \mathcal{F}}(\mathbf{pk}, \mathbf{vk})$</u>	<u>$\text{Ideal}_{\mathcal{A}, n, d, K, h, \text{seed}}(\mathbf{pk}, \mathbf{vk}, \text{trap})$</u>
1. $C_{\mathcal{F}} \leftarrow \text{ComForest}(\mathbf{pk}, \mathcal{F}, r_F)$	1. $C_{\mathcal{F}} \leftarrow S_1(\mathbf{pk}, \text{trap}, h, K)$
2. $\mathbf{x} \leftarrow \mathcal{A}(\mathbf{vk}, C_{\mathcal{F}}, h, K)$	2. $\mathbf{x} \leftarrow \mathcal{A}(\mathbf{vk}, C_{\mathcal{F}}, h, K)$
3. $\pi_P, y \leftarrow \text{ProvePred}(\mathbf{pk}, \mathcal{F}, r_F, \mathbf{x})$	3. $\pi_P, y \leftarrow S_2(\mathbf{pk}, \text{trap}, C_{\mathcal{F}}, \mathbf{x}, h, K)$
4. $b \leftarrow \mathcal{A}(\mathbf{vk}, C_{\mathcal{F}}, \pi_P, \mathbf{x}, y, h, K)$	4. $b \leftarrow \mathcal{A}(\mathbf{vk}, C_{\mathcal{F}}, \pi_P, \mathbf{x}, y, h, K)$
5. Return b	5. Return b

Now we have everything we need to present *zkFTP*. First, in section 4.1 we introduce a certification algorithm for checking the correctness of training of decision trees and forests which is asymptotically faster than training the model from scratch. In subsequent sections, we present our algorithms with emphasis on efficiently proving the correct training of a forest by utilizing our certification algorithm. To simplify the presentation, in section 4.2, we focus on *zkFTP* for a single tree \mathcal{T} (i.e., $K = 1$). Next, in section 4.3, we show how to generalize our construction for forests ($K > 1$).

4.1 Certifying Correctness of Decision Trees

Tree training requires building the tree by repeated splits of the dataset per tree node. Asymptotically, this takes $\mathcal{O}(h|D|)$. Moreover, it entails mostly comparisons, as well as other components like recursive tree traversal and random memory accesses, all of which are “costly” to compile inside a circuit-based zkSNARK. Motivated by this, we propose a lightweight certification algorithm which, given as input a tree \mathcal{T} and a dataset D , validates the correctness of the training of the first without having to redo it. At a high level, it computes the histograms of the tree leaves and, based on their homomorphic properties, calculates the histograms of the remaining nodes. Then, it uses the latter to validate the correctness of node splits. In more detail, our certification algorithm initially checks that \mathcal{T} is a well-formed tree, and then carries out the following steps:

1. First, we assign each element to its corresponding leaf. Namely, let $P : \mathbb{N} \rightarrow [L]$ be an assignment function, where L the number of leaves, then for every point $p_j \in D$, perform an inference step and, depending on which leaf it lies in, we update $P[j]$ accordingly (e.g., if $p_j \in D$ lies in the i -th leaf of the tree, then $P[j] = i$).

2. In the second step, and using P , we compute the histograms for each leaf, feature $j \in [d]$ and label. In the end, for every partition, we computed $2d$ histograms of size B each.
3. In the third step, we compute the histograms of the non-leaf nodes. We observe that histograms are homomorphic, in the sense that if H_{v_l} and H_{v_r} are the histograms of the children of a node v , then $H_v = H_{v_l} + H_{v_r}$. Thus, starting from leaves, we aggregate the histograms until reaching the root.
4. Starting from the root, use each node’s histograms to compute the best-split value and its feature and compare them with the corresponding node in \mathcal{T} . If they are not the same, reject, otherwise proceed to its children. When checking a leaf, additionally check if the stopping criteria are met (see Section 2).

Algorithm 1 gives a detailed description of our tree certification algorithm, denoted as `CERTIFYTREE`. It should be noted that `CERTIFYTREE` takes as input the tree \mathcal{T} , dataset D , and a vector $\mathbf{F}_B \in \mathbb{N}^n$ (referred to as a frequency vector), which will be utilized later during forest training certification. Initially, we assume that \mathbf{F}_B is an n -sized vector consisting of ones.

Having established that, the algorithm starts by initializing the partition assignment array P , the leaf histograms H^{leaf} , and a list \mathcal{N} containing all the tree \mathcal{T} ’s leaves (line 12). It then progresses to the first step (lines 13-15), where it iterates over the dataset once, invoking the `TREEPREDICT` algorithm for each data point to obtain the predicted leaf’s ID i and updating P accordingly. Subsequently, using the vector P , the algorithm advances to the second phase by computing the leaf histograms (lines 16-17). After this computation, it calculates histograms for the remaining nodes (lines 18-24) as follows: Initially, it sets up a dictionary H that maps tree nodes to their respective histograms (lines 18-19). It then retrieves the sibling of the first node in \mathcal{N} (line 21), removes both nodes from \mathcal{N} and adds their parent (line 23), updating the dictionary by mapping the parent node to the sum of the children’s histograms (line 24). Finally, the algorithm transitions to the last phase, where it verifies the correctness of splits by employing `VALIDATESPLIT`. Broadly speaking, `VALIDATESPLIT` follows a structure akin to `FINDSPLIT` from training algorithm 3, but it leverages the previously computed histograms and focuses solely on assessing the optimal split.

The proof of the correctness of our certification algorithm is in Appendix C.1. At a high level, if \mathcal{T} is invalid, the algorithm will always reject. For instance, if the root of \mathcal{T} is incorrect (e.g., wrong split-value or feature), our algorithm will detect this with probability one. That derives from the fact that the root histograms we computed at step (3) are identical to the histograms the training algorithm computes to perform the first split. Consequently, our algorithm computes the same root node as the “honest” training algorithm. Next, if the split is valid, we know that the histograms of the root’s children are correct. That holds because, at step (1), data assignment to leaves depends on the correctly computed root. The same argument applies recursively for the rest of the nodes. We note our certification assumes a deterministic `TRAIN`; in practice, this is not a limitation since a randomized training algorithm can be “de-randomized” by explicitly providing the random *seed* as input.

Regarding performance, the computation of assignments and leaf histograms requires one scan of the dataset and $\mathcal{O}(|D|)$ steps (“on the fly” without having to store P). Afterwards, steps 3-4 take $\mathcal{O}(2d|\mathcal{T}|B)$. For all practical purposes, $2d|\mathcal{T}|B < |D|$ [78, 75], leading to an overall complexity of $\mathcal{O}(|D|)$. This is a big improvement over the $\mathcal{O}(h|D|)$ required by the training algorithm! Besides, the certification algorithm needs space $|D|$ (for the dataset), $|\mathcal{T}|$ (for the tree), and $2 \cdot d \cdot |\mathcal{T}| \cdot B$ (for the histograms), totaling to $\mathcal{O}(|D| + d|\mathcal{T}|B)$.

Extending to Forests. Using the tree certification algorithm, we can construct an algorithm that certifies the correctness of the training of a forest. We provide a detailed description of the

Algorithm 1 Tree Certification Algorithm

```
1: procedure VALIDATESPLIT( $\mathcal{T}, H, node, \mathcal{A}$ )
2:   if ISLEAF( $\mathcal{T}, node$ ) = 1 then return 1
3:    $g \leftarrow \mathbf{0}^{|\mathcal{A}|}$ 
4:   for all  $j \in \mathcal{A}$  do
5:      $\epsilon \leftarrow \epsilon \cup (j, \text{BESTSPLITSORE}(Hist_{node,j,0}, Hist_{node,j,1}))$ 
6:    $v, j = \text{MIN}(\epsilon)$ 
7:   if  $v \neq node.v \wedge j \neq node.a$  then return 0
8:    $s_1 \leftarrow \text{VALIDATESPLIT}(\mathcal{T}, H, node.left, \mathcal{A} - \{j\})$ 
9:    $s_2 \leftarrow \text{VALIDATESPLIT}(\mathcal{T}, H, node.right, \mathcal{A} - \{j\})$ 
10:  return  $s_1 \wedge s_2$ 
11: procedure CERTIFYTREE( $\mathcal{T}, D, \mathbf{F}_B$ )
12:  Set  $P = \mathbf{0}^n, H_{i,j,k}^{leaf} \leftarrow \mathbf{0}^B, \forall (i, j, k) \in (L, [d], [2]), \mathcal{N} \leftarrow \mathcal{L}$ 
13:  for all  $j \in [n]$  do
14:     $i, _ \leftarrow \text{TREEPREDICT}(\mathcal{T}, D_j)$ 
15:     $P[j] = i$ 
16:  for all  $i \in [n], j \in [d]$  do
17:     $H_{P[i],j,D_i.label}^{leaf}[D_i[j]] = H_{P[i],j,D_i.label}^{leaf}[D_i[j]] + \mathbf{F}_B[i]$ 
18:  for all  $i \in \mathcal{N}$  do
19:     $H[i] \leftarrow H_i^{leaf}$ 
20:  while  $|\mathcal{N}| \neq 1$  do
21:     $i_1, i_2 \leftarrow \text{GETSIBLINGS}(\mathcal{N})$ 
22:     $i \leftarrow \text{GETPARENT}(\mathcal{T}, i_1, i_2)$ 
23:     $\mathcal{N} \leftarrow \mathcal{N} \cup \{i\} - \{i_1, i_2\}$ 
24:     $H[i] \leftarrow \{H_{i_1, \cdot, \cdot} + H_{i_2, \cdot, \cdot}\}$ 
25:  return  $\text{VALIDATESPLIT}(\mathcal{T}, H, \mathcal{T}.root, [d])$ 
```

certification process in Algorithm 2. More precisely, given as input the forest \mathcal{F} , dataset D , the number of trees K and a seed needed to generate randomness, CERTIFYFOREST iterates over every tree \mathcal{T}_i of \mathcal{F} , computes the frequency of each data point in the bagged dataset (used to train \mathcal{T}_i) by invoking the GETFREQUENCIES algorithm and runs the Algorithm 1 with input the tree \mathcal{T}_i , dataset D and frequency vector \mathbf{F}_i .

Note that correctness stems directly from the correctness of the certification algorithm for a single tree. As for complexity, line 4 requires $\mathcal{O}(n)$ steps and line 5 $\mathcal{O}(|D|)$ steps. So the total computational complexity of algorithm 2 is $\mathcal{O}(K|D|)$. Finally, space complexity equals to the space needed to execute one invocation of the tree certification algorithm plus $|\mathcal{F}|$.

4.2 Our *zkFTP* Construction for Trees

Here we present our construction for zero-knowledge forest training and prediction for a single tree, with an emphasis on tree training. In the next section we will show how to generalize our techniques for forests.

Key Generation & Commitment. Given the dataset dimensions n, d , *KeyGen* invokes the key generation algorithm of our space-efficient PC scheme for polynomials of size nd . As we will explain below, the prediction phase requires encoding the tree as a univariate polynomial, hence we also

Algorithm 2 Forest Certification Algorithm

```

1: procedure CERTIFYFOREST( $\mathcal{F}, D, K, seed$ )
2:    $b \leftarrow 1$ 
3:   for  $i \in [K]$  do
4:      $\mathbf{F}_i, seed \leftarrow$  GETFREQUENCIES( $D, seed$ )
5:      $b \leftarrow b \wedge$  CERTIFYTREE( $\mathcal{T}_i, D, \mathbf{F}_i$ )
6:   return  $b$ 
  
```

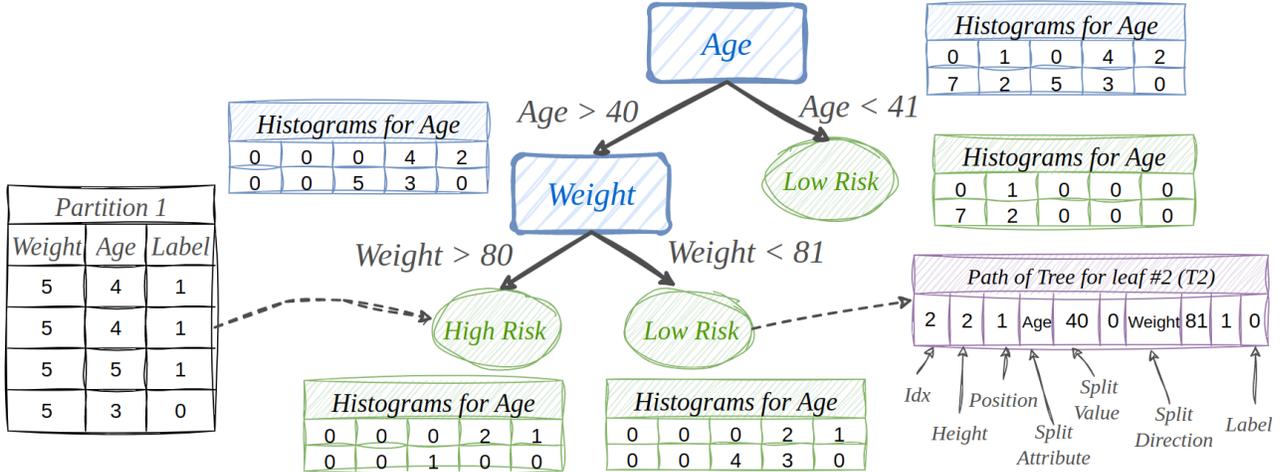


Figure 1: Detailed description of the main data structures for training with max bin size $B = 5$. Leaf histograms for Age are in green tables, and non-leaf histograms in blue ones. The purple table shows the path from the root to the second leaf of the tree.

generate parameters for the univariate KZG PC [27]. *ComData* commits to the multi-linear extension of D again using our space-efficient PC. For *ComForest*, we first encode \mathcal{T} as a matrix $T \in \mathbb{F}^{L \times p}$, where each row, corresponds to a leaf, containing its id, coordinates (height and relative position) and path to the root (see Figure 1). We commit to \mathcal{T} by computing the polynomial commitment of the multi-linear extension of T via our PC scheme.

Proving Training. For dataset D and tree \mathcal{T} , our *ProveTrain* generates a zero-knowledge proof of training by showing that “the certification algorithm on input D and \mathcal{T} accepts”. We follow a modular approach, proving each step of the certification algorithm independently by first translating it into a data-parallel arithmetic circuit that performs the corresponding checks, instantiating the streaming oracle for its input, and then invoking SPARROW. In this way, we can also utilize specialized protocols that solely rely on the sumcheck (e.g., multiplication trees of [67] or the range proofs of [7]) and result in concretely faster proving times. In addition, following this approach, we can also pre-process parts of the proof that do not explicitly depend on the training process (e.g., proving the correctness of bagging), leading to faster “online” proving times. Here, we provide a high-level description of the circuit and leave correctness proofs and details on how to express permutation and multi-set checks into arithmetic circuits in Appendix C.3.

(1) Checking Correctness of Assignments. Let $T' \in \mathbb{F}^{n \times p}$ be an auxiliary matrix which, for every data point, contains its path in the tree (e.g., inference of point $p_i \in D$, will follow the T'_i path). Because every path encodes the leaf index (leftmost entry of purple table of Figure 1), to prove the correct assignment of every point in a leaf, it is enough to prove T' is “well-formed” by showing that

(i) $\forall p_i \in D$, T'_i is the correct path of p_i (ii) all paths from T' are rows of T .

Check (i) is done with a data-parallel circuit (denoted with \tilde{C}_1) where each sub-circuit takes as input a point p_i and path T'_i and validates the correctness of inference. For this we use the circuit of [39]. For check (ii), we need to prove that $\{T'_i\}_{i \in [n]} \subseteq \{T_i\}_{i \in [L]}$ using the “standard” multi-set check [28, 39], realized via two multiplication tree circuits of size $\mathcal{O}(D)$ and $\mathcal{O}(|T|)$ (denoted with \tilde{C}_2 and \tilde{C}_3 respectively). Since this last circuit has size $\mathcal{O}(|T|)$ we can directly invoke the standard GKR-based zkSNARK [10] for it. The first two have size $\mathcal{O}(|D|)$ but since they are data-parallel, we can directly use SPARROW for them.

At this point, recall that to use SPARROW, we need to instantiate streaming access to the input of the corresponding circuit. To achieve that, we first instantiate streaming access to T' by creating a routine which, on the j -th invocation, uses $S(D)$ to get the next point p_j and performs an inference step to compute and output T'_j . Using $S(T'_j)$, we can construct the streaming oracles for the inputs of the circuits \tilde{C}_1 and \tilde{C}_2 . For the first circuit, observe that in the j -th invocation, we need to output the tuple (p_j, T'_j, aux_j) , where aux_j is auxiliary information required for the prediction circuit (e.g., bit-decomposition values). We do so by first invoking $S(D)$ to get p_j , $S(T'_j)$ to get T'_j , generate aux_j and output (p_j, T'_j, aux_j) . As for the second circuit, we only need to use $S(T')$.

(2) Checking Leaf Histograms. Let us first focus on a single histogram. To check its correctness, we create a data-parallel arithmetic circuit that takes $\mathbf{h} \in [B]^N$ and outputs a histogram $H \in \mathbb{F}^B$, such that $H[i] = \sum_{j \in [N]} I(\mathbf{h}_j = i)$. The “straightforward” way, as proposed in [87], is to represent each element h_i , via its one-hot encoding $\rho_{h_i} \in \{0, 1\}^B$, and aggregate all encodings to compute H . Unfortunately, this results in a circuit of size $\mathcal{O}(BN)$. We significantly reduce this to $\mathcal{O}(N)$, by treating the histogram H as a random access memory, with address space all values in $[B]$. Thus, we can compute H by scanning \mathbf{h} , updating $H[\mathbf{h}_i]$ to $H[\mathbf{h}_i] + 1$ for every $i \in [N]$. This can now be proven efficiently via offline memory consistency checks [88, 11, 89]. In particular, given \mathbf{h} , the initialized histogram $H_b = \{(i, 0), \forall i \in [B]\}$, the computed histogram $H = \{(i, H[i]), \forall i \in [B]\}$ and a memory read/write transcript $H_r = \{(\mathbf{h}_i, v_i)\}_{i \in [N]}$, $H_w = \{(\mathbf{h}_i, v_i + 1)\}_{i \in [N]}$, we must ensure that (i) every i -th pair in H_r, H_w has the correct address and updated value, and (ii) H_r is a permutation of a consistent read transcript in which a pair (i, v_i) exists if and only if the pair (i, v_i) already belongs to H_w . For (i) we use a data-parallel circuit (denoted with \tilde{C}_4) that takes as input H_r, H_w, \mathbf{h} , and for each $i \in [N]$ checks if $H_r[i][0] = H_w[i][0] = \mathbf{h}_i$ and $H_r[i][1] + 1 = H_w[i][1]$. For (ii), we rely on the observations of [88], showing that the read/write transcripts are consistent if $H \cup H_r = H_b \cup H_w$. Using the standard techniques of [11, 39, 28], we prove that $H \cup H_r = H_b \cup H_w$ using a single circuit (denoted with \tilde{C}_5) that encodes four multiplication trees.

To apply this to the histograms of all leaves we extend the address space from a single value to the tuple $(j, v_j, i, \{0, 1\})$ where $j \in [d]$ is the feature, $v_j \in [B]$ is the value of a data point at j , $i \in [L]$ the leaf index and $\{0, 1\}$ the label of the data point. Because both circuits are data-parallel we can use SPARROW to prove their correct computation.

What remains is to show how to instantiate streaming access to \tilde{C}_4 and \tilde{C}_5 . To begin with, given streaming access to the dataset D and T' , we can instantiate $S(\mathbf{h})$ in the following way. On the j -th invocation, we invoke $S(D), S(T')$, receive $p_j, y_j = p_j.label, T'_j$ and output $\{h_{i,j}\}_{i \in [d]} = \{(i, p_{j,i}, T'_j[0], y_j)\}_{i \in [d]}$, where $T'_j[0]$ stores the leaf index of p_j . For $S(H_r), S(H_w)$, we use $S(\mathbf{h})$ and an auxiliary buffer $H \in \mathbb{N}^B$. Specifically, upon the j -th invocation of H_r (and H_w resp.) we invoke $S(\mathbf{h})$ to get $\{h_{i,j}\}_{i \in [d]}$ return $\{h_{i,j}, B[h_{i,j}]\}_{i \in [d]}$ (and $\{(h_{i,j}, B[h_{i,j}] + 1)\}_{i \in [d]}$ resp.) and increment $B[h_{i,j}]$ by one. Similarly, we can also compute the final histograms H^L , which we store locally. Having established that, we can use $S(H_w), (H_r)$ and $S(\mathbf{h})$ to instantiate streaming access to the input of \tilde{C}_4 and $S(H_w), (H_r), H^L$ we can instantiate streaming access to the input of \tilde{C}_5 .

(3) Checking Non-Leaf Histograms. For the third step, we must prove the correctness of non-leaf

histograms. More generally, we can prove *computations over trees*. Formally, we want to create a circuit that, given as input the leaves, their data (in our case, the histograms), and a function $\mathcal{G} : \mathbb{F}^{|D_{vr}|+|D_{vl}|} \rightarrow \mathbb{F}^{|D_v|}$ applied to the data, outputs the data of all non-leaf nodes of the tree. A direct approach would be to “hardwire” the tree topology in the circuit. Unfortunately, the topology not only is unknown a priori but may also need to remain hidden from the verifier. Alternatively, we can enforce all leaves to have the same height (i.e., by adding padding nodes/leaves), but the circuit would be of size at least $\mathcal{O}(2^{h_{max}})$, where h_{max} is the maximum tree depth, leading to considerable overhead, e.g., when the tree is unbalanced. Next, we present a circuit whose size is only linear to the number of nodes, at the cost of revealing to the verifier an upper bound on the number of nodes.

First, let \mathcal{L} be the set of pairs containing leaf coordinates denoted as $coord(v) = (h, p)$ (h is height and p is relative position in the tree) and their data D_v . Furthermore, let \mathcal{S} be the computation transcript of the tree, i.e., a set containing the coordinates of all sibling nodes of the tree and their corresponding data (if $\mathcal{S}_i = \{(coord(v_{i,l}), D_{v_{i,l}}), (coord(v_{i,r}), D_{v_{i,r}})\}$ is such a pair then $\mathcal{S} = \bigcup_{i \in [|T|]} \mathcal{S}_i$). Given these sets, we generate a circuit that first checks that every element \mathcal{S}_i encodes two siblings (e.g., $p_l = p_r + 1$, $h_l = h_r$) and then outputs the coefficients and data of their parent as $(h_l - 1, p_r/2, \mathcal{G}(D_1, D_2))$. Let \mathcal{N} be the set containing all output pairs. Note that, if \mathcal{S} corresponds to a correct transcript of a tree computation, then \mathcal{N} is the set containing the coordinates of every non-leaf node and its data. Thus, we test whether $\mathcal{N} \cup \mathcal{L} = \mathcal{S} \cup (0, 0, D_r)$, again using a multi-set check, where $(0, 0, D_r)$ corresponds to the coordinates and data of the root.

To prove the correctness of histograms H_N , we use this circuit with leaf data being their histograms and function \mathcal{G} vector addition. It also receives the tree to retrieve leaf coordinates. The circuit has $\mathcal{O}(dLB)$ size so we again use the standard GKR-based zkSNARK.

(4) Checking Node Splits. This uses a circuit (denoted with $\tilde{\mathcal{C}}_6$) that validates the correctness of the splits by computing the Gini index [72] for each entry of all histograms (checked in the previous step), comparing them with the optimal ones. Even though this circuit has size $\mathcal{O}(dLB)$, it entails a large multiplicative overhead, as it needs multiple comparisons (e.g., using range proofs). Hence, we prove its correct evaluation circuit using SPARROW to reduce space complexity.

(5) Checking Tree Well-Formedness. Finally, we need to ensure that T corresponds to a tree graph and there are no inconsistencies between paths, i.e., all sibling leaves/nodes follow the same path to the root. We can check this with a circuit (denoted with $\tilde{\mathcal{C}}_7$) for computations over trees where function \mathcal{G} takes the paths of two sibling nodes and checks if their paths to the root are the same. This circuit has size $\mathcal{O}(|T|)$ and we use the standard GKR-based zkSNARK.

Verifying Training. Finally, the verifier receives five proofs and corresponding commitments, referring to steps (1)-(5) above, and checks them with the verification algorithms of the standard GKR-based SNARK or SPARROW, accordingly. Note that these proofs refer to computations performed over have to share some common inputs, i.e., steps (1),(2) share T' , (2),(3) the leaf histograms, and (3),(4) the non-leaf histograms. The verifier “glues” the verification process together by utilizing the same corresponding commitments for these inputs when validating them.

Proving & Verifying Predictions. Having established the correctness of the model, the generation of zero-knowledge proof of prediction for a given test point works as follows. First, it commits to the path $\mathbf{p} \in \mathbb{F}^p$ the prediction algorithm follows. Next, it uses the matrix lookup argument of [40] to prove that \mathbf{p} is a row of T . Finally, to validate the correctness of \mathbf{p} for the test point, it invokes the prover for the prediction circuit of [39] for \mathbf{p} , \mathbf{x} and label y . Due to the performance guarantees of matrix lookup arguments, the proving time will be independent of the forest size.

Unfortunately, this has a caveat. All existing matrix lookup arguments [40, 90] assume the data was committed with a univariate PC scheme, while our scheme relies on multi-variate polynomial encodings. To overcome this, in *ComForest*, the prover uses a univariate PC scheme [27] to commit

to $p(x) = \sum_{i,j \in [L] \times [p]} L_i(x)T_{i,j}$, which is univariate polynomial encoding of T . It then proves that the coefficients of p and f_T (where the latter is the multi-linear extension of T that we previously committed to) are the same by choosing a random point s , generating an evaluation proof for $p(s)$, and proving the sumcheck instance $p(s) = \sum_{x \in \{0,1\}^{\log Lp}} f_{\mathbf{y}}(x)f_T(x)$, where $\mathbf{y} = (L_1(s), \dots, L_{Lp}(s))$.

4.3 Extending to Random Forests

Given the baseline case where $K = 1$, we can extend our *zkFTP* scheme to support forests (i.e., when $K > 1$). In this section, we will outline the necessary modifications required in all algorithms.

Adapting Key Generation & Commitment: *KeyGen* takes as an additional input the maximum number of trees in the forest K and invokes the key generation algorithm of our space-efficient PC scheme for polynomials of size ndK . For *ComForest*, observe that a forest \mathcal{F} , is a set of K trees $\{\mathcal{T}_1, \dots, \mathcal{T}_K\}$. Based on this, we encode \mathcal{F} using a matrix $F \in \mathbb{F}^{KL \times p}$, consisting of the concatenation of the tree matrixes T_i as defined in section 4.2. Furthermore, we slightly update $T_i \in \mathbb{F}^{L \times p}$ by inserting an additional entry on every row containing the index of the tree (i.e., i). We will use the latter entry to validate that the prover selected one row from every tree. Similarly, we commit the univariate polynomial representation of F . Finally, *ComData* remains unchanged.

Adapting Proving Training: As in the single tree case, we prove the training of a random forest by following a modular approach, i.e., independently proving each step of the certification algorithm. Recall that the forest certification algorithm consists of K invocations of the tree certification algorithm with an additional step of computing the frequency vector for each tree. Consequently, to prove the correctness of forest training, it is enough to adapt steps (1)-(5) to work over K trees and incorporate a new step for proving the bagging process for each tree.

(0) Checking Bagging Correctness. In this step, instead of proving the correct computation of the bagged dataset D_i , we compute the frequency of every data point on D_i . Because we sample points uniformly at random and with replacement, the frequency of a data point at any D_i follows a binomial distribution. More precisely, assuming that $\mathbf{F}_B \in \mathbb{N}^{K \times n}$ be the matrix that contains the frequency of every data point on every tree, we need to prove that every element of \mathbf{F}_B follows a binomial distribution with n number of trials and $1/n$ probability of success.

To achieve that, at a high level, we need to prove the correct computation of a circuit that takes as input a seed (selected by a verifier), a small pre-agreed table T_r (e.g., of size 2^{16} elements and following binomial distribution) and for each $i \in [Kn]$, (i) generates a number \mathbf{g}_i uniformly at random (ii) outputs $\mathbf{F}_B[i] = T_r[\mathbf{g}_i]$. For the first part, we use the Linear congruential generator [91], which generates \mathbf{g}_i by computing $\mathbf{g}_i = (A\mathbf{g}_{i-1} + B) \text{mod} |T_r|$, for A, B public parameters. So, to prove (i), we will create a circuit (denoted with $\mathcal{C}_{B,1}$) that takes as input all \mathbf{g}_i 's and validates their correctness. Note this circuit is data parallel with constant depth as it consists of kN sub-circuits which contain two range proofs (for the modulo operation), two additions, and one differentiation (to ensure that the input of \mathbf{g}_i is the output of the $i - 1$ -th invocation). For (ii), we need to show that $\{(\mathbf{g}_i, \mathbf{F}_B[i])\}_{i \in [Kn]}$ is a multi-set of $\{(i, T_r[i])\}$, which is done by using a circuit (denoted with $\mathcal{C}_{B,2}$) that computes a multiplication tree. Because both circuits are data-parallel, we can use SPARROW to prove their correct computation.

What remains is to show how to instantiate streaming access to the input of both circuits. We begin by showing how to instantiate $S(\mathbf{g}), S(\mathbf{F}_B)$, where $\mathbf{g} \in \mathbb{F}^{nK}$ is the randomness vector and $\mathbf{F}_B \in \mathbb{F}^{nK}$ is the frequencies vector. For the first one, observe that $\mathbf{g}_i = (A\mathbf{g}_i + B) \text{mod} |T_r|$. Due to the incremental nature of the computation, we can naturally instantiate streaming access to \mathbf{g} with minimal space overhead. Specifically, to instantiate $S(\mathbf{g})$, we only need to maintain the element of the previous invocation and $A, B, |T_r|$. Given $S(\mathbf{g})$, we can construct $S(\mathbf{F}_B)$ by invoking $S(\mathbf{g})$ to get the next randomness \mathbf{g}_i and outputting $T_r[\mathbf{g}_i]$.

To generate streaming access to the input of $\mathcal{C}_{B,1}$, on the j -th invocation, we invoke $S(\mathbf{g})$ to get \mathbf{g}_j , and compute the auxiliary information (e.g., bit values) to perform the modulo operation. For $\mathcal{C}_{B,2}$, we first use $S(\mathbf{g}), S(\mathbf{F}_B)$ to compute and locally store \mathbf{w}_3 . We can access the rest of the input elements using $S(\mathbf{g}), S(\mathbf{F}_B)$.

(1) Checking Correctness of Assignments. We modify this step by proving the correct computation of the circuits $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ consisting of K identical copies of the sub-circuits $\tilde{\mathcal{C}}_1, \tilde{\mathcal{C}}_2, \tilde{\mathcal{C}}_3$. Because $|\mathcal{C}_3| = \mathcal{O}(|\mathcal{F}|)$, we can prove its correct computation by invoking the standard GKR-based zkSNARK. For \mathcal{C}_1 and \mathcal{C}_2 we use SPARROW. Likewise, we modify the streaming oracles of the inputs of $\mathcal{C}_1, \mathcal{C}_2$ by constructing a wrapper that invokes K streaming oracles, one for each tree, in sequential order, in the sense that we invoke the i -th oracle after scanning all the data of the $i - 1$ -th oracle. For instance, in the j -th invocation, $S(T')$ (where $T' = T^1 | \dots | T^K$), will invoke $S(T'^{j/(n \cdot p)})$ and output $T'^{j/(n \cdot p)}_{[j]_{(n \cdot p)}}$.

(2) Checking Leaf Histograms. Similarly to the previous step, we prove the correct computation of the circuits $\mathcal{C}_4, \mathcal{C}_5$ consisting of K identical copies of the sub-circuits $\tilde{\mathcal{C}}_4, \tilde{\mathcal{C}}_5$. Note, however, that we need to slightly modify $\tilde{\mathcal{C}}_4$ in the following way. The i -th copy $\tilde{\mathcal{C}}_4$ takes as an additional input the vector $\mathbf{F}_B[i \cdot n : (i + 1)n]$ and instead of incrementing each bin by one (when accesses it), updates it based on the input frequency. Because $\mathcal{C}_4, \mathcal{C}_5$ are data-parallel circuits, we use SPARROW to prove their correct computation. Similarly, with the streaming oracles of **Step 2**, we can use the same compiler to instantiate $S(\mathbf{h}), S(H_r), S(H_w)$. In addition, since \mathcal{C}_4 takes as input \mathbf{F}_B , we also need to use $S(\mathbf{F}_B)$.

(3) Checking Non-Leaf Histograms & (4) Proving Splits. We adapt these steps by using the circuits \mathcal{C}_6 and \mathcal{C}_7 respectively, consisting of K copies of the sub-circuits $\tilde{\mathcal{C}}_6$ and $\tilde{\mathcal{C}}_7$. To prove the correct computation of both circuits, we use SPARROW. Like in the previous steps, we can instantiate streaming access to their inputs.

Adapting Training Verification. Training verification remains almost unchanged, with the only exception that the verifier needs to validate the correctness of the proof generated in step (0). Furthermore, it has to link the commitment of \mathbf{F}_B with step (2).

Adapting Proving & Verifying Predictions: We can extend the protocol of Section 4.2 for a random forest by selecting the correct paths for \mathbf{x} , one for each tree in the forest, denoted with $\mathbf{P} \in \mathbb{F}^{K \times p}$. Then, invoke the matrix lookup argument to prove that every row of \mathbf{P} belongs to F . Finally, prove the correct computation of the prediction circuit but extended for forests (see [39]) for more details. As for the protocol that links the pre-image of the univariate with the multi-variate commitment of F , it remains unchanged.

Construction 2 gives a detailed description of our *zkFTP* scheme. For the matrix lookup arguments, we make black-box use of the definition provided by [40]. Specifically, the matrix lookup argument consists of four algorithms (*Derive, Preproc, Prove, Verify*). At a high level, *Derive* takes as input univariate KZG parameters and outputs specialized public parameters necessary for proving the matrix lookup argument. *Preproc*, is used by the prover to generate some auxiliary information used to improve the complexity, *Prove* generates a zero-knowledge proof $\pi_{l_{kp}}$ for $\mathcal{R}_{l_{kp}} = \{(C_{\mathcal{F}}, C_{\mathbf{P}}, K, L, p); (\mathbf{P} \in \mathbb{F}^{K \times p}, \mathbf{F} \in \mathbb{F}^{K \times L \times p}) : \mathbf{P} \subseteq \mathbf{F}\}$ and *Verify* validates it. Note that *ProveTrain* invokes **Construction 3** in which we present our proof of training protocol for forests. Finally, we can claim the following, which we prove in Appendix C:

Theorem 3 *Our construction is a zkFTP with proving complexity of $\mathcal{O}(|D|K \log \log K|D|)$, $\mathcal{O}(|D| + |\mathcal{F}| + LBd + \sqrt{|D|K})$ space complexity, and $\mathcal{O}(\log K|D|)$ proof size and verification time for training. For prediction, it has $\mathcal{O}(hK \log hK)$ prover time, $\mathcal{O}(d)$ verification and $\mathcal{O}(1)$ proof size.*

Construction 2 (Zero-Knowledge proofs of Forest Training and Predictions): Let λ be the a security parameter, n be the maximum number of points, d their dimension, h the maximum depth of the trees, K the maximum number of trees and seed a random element used to bootstrap randomness.

- **KeyGen**($1^\lambda, n, d, h, K$): Invoke $\mathbf{pk}_T, \mathbf{vk}_T \leftarrow PC.KeyGen(1^\lambda, \log(ndK), 1)$ to generate the public parameters for committing the data, proving and verifying the forest training. Invoke the univariate KZG key generation $\mathbf{pk}_{kzg}, \mathbf{vk}_{kzg} \leftarrow KZG.KeyGen(1^\lambda, \kappa(d, h, K))$, $\mathbf{pk}_P, \mathbf{vk}_P \leftarrow MLookup.Derive(srs, K2^h, p, K)$ to generate the public parameters for proving and verifying prediction (for more details on *MLookup*, see Figure 2, in Section 5 of [40]). Finally return $(\mathbf{pk} = \{\mathbf{pk}_T, \mathbf{pk}_{kzg}, \mathbf{pk}_P\}, \mathbf{vk} = \{\mathbf{vk}_T, \mathbf{vk}_{kzg}, \mathbf{vk}_P\})$.
- **ComData**(\mathbf{pk}, D, r_D): Invoke $C_D \leftarrow PC.Commit(\mathbf{pk}_T, f_D, r_D)$ and return C_D .
- **ComForest**($\mathbf{pk}, \mathcal{F}, r_F$): Encode \mathcal{F} to a matrix $F = ((T_1)^T | \dots | (T_K)^T)^T \in \mathbb{F}^{L \times p}$, as defined in Section 4.3, where L is the maximum number of leaf nodes of the trees and $T_i \in \mathbb{F}^{L \times p}$ is a matrix representation of a tree as explained in Section 4.2. Moreover, let g_F be the multi-linear extension of F and p_F the univariate polynomial defined as $p_F(x) = \sum_{i,j \in [KL] \times [p]} L_{bi+j}(x) F_{i,j}$. Invoke $C_{\mathcal{F}}^T \leftarrow PC.Commit(\mathbf{pk}_T, g_F, r_F^{(1)})$ and $C_{\mathcal{F}}^P \leftarrow KZG.Commit(\mathbf{pk}_{kzg}, p_F(x), r_F^{(2)})$. Return $C_{\mathcal{F}} = \{C_{\mathcal{F}}^T, C_{\mathcal{F}}^P\}$.
- **ProveTrain**($\mathbf{pk}, D, \mathcal{F}, r_D, r_F, seed$):
 1. **Prove the correctness of training:** Invoke **Construction 3**, and generate ten proofs (π_1, \dots, π_{10}) and the commitments $C_{F_B}, C_{T'}, C_{H^L}, C_H$.
 2. Return $\pi_T = \{\{C_{F_B}, C_{T'}, C_{H^L}, C_H\}, \{\pi_1, \dots, \pi_{10}\}\}$
- **VerifyTrain**($\mathbf{vk}, C_D, C_{\mathcal{F}}, \pi_T, seed$):
 1. Verify bagging using $\pi_1, \pi_2, C_{F_B}, seed$ and T_r .
 2. Verify the correct partitioning of data using $\pi_3, \pi_4, \pi_5, C_{T'}, C_D$ and $C_{\mathcal{F}}$.
 3. Verify the correct computation of histograms using $\pi_6, \pi_7, C_{T'}, C_{F_B}$ and C_{H^L} .
 4. Verify the correct computation of non-leaf histograms using π_8, C_{H^L}, C_H and $C_{\mathcal{F}}$.
 5. Verify the correctness of splits using π_9, C_H and $C_{\mathcal{F}}$.
 6. Verify tree well-formedness using π_{10} and $C_{\mathcal{F}}$.
- **ProvePred**($\mathbf{pk}, \mathcal{F}, r_F, \mathbf{x}$):
 1. If \mathcal{P} invokes this method for the first time it must compute the following additional proofs:
 - (a) Call $aux \leftarrow MLookup.Preproc((\mathbf{pk}_{kzg}, \mathbf{vk}_{kzg}), F)$ and locally store aux .
 - (b) **Prove that the pre-images of $C_{\mathcal{F}}^T$ and $C_{\mathcal{F}}^P$ are the same:** Pick a random point $s \in \mathbb{F}$. Generate a KZG evaluation proof for p_F at s . Next, assuming that $\mathbf{y} = (L_1(s), \dots, L_{KLp}(s))$, prove the sumcheck instance $p_F(s) = \sum_{x \in \{0,1\}^{\log KLp}} f_{\mathbf{y}}(x) g_F(x)$. Finally, generate an evaluation proof for the claim of g_F derived by the sumcheck protocol and prove the correct computation of $f_{\mathbf{y}}(s)$ (using the same techniques as with our space-efficient sumcheck). Set π_{link} the proof, that internally contains all the PC evaluation proofs and sumcheck proofs.
 2. Let $\mathbf{P} \in \mathbb{F}^{K \times p}$, be a matrix containing K paths (one for each tree) \mathbf{x} follows when making a prediction. Let $\mathbf{p} \in \mathbb{F}^{Kp}$ be the vectorization of \mathbf{P} . Compute and send to \mathcal{V} the commitment $C_p \leftarrow KZG.Commit(\mathbf{pk}_{kzg}, p_p(x), r_p)$, where $p_p = \sum_{i \in [Kp]} L_i(x) \mathbf{p}_i$.
 3. Invoke $MLookup.Prove(\mathbf{pk}_P, (C_{\mathcal{F}}^P, C_p), \mathbf{P}, aux)$ and generate a proof π_{lkp} , proving that every row in \mathbf{P} is a row in F^T .
 4. Let $\mathcal{R}_{pred} = \{(C_p, \mathbf{x}, \mathbf{y}); \mathbf{p}, \mathbf{w} : \mathcal{C}_{pred}(\mathbf{p}, \mathbf{w}, \mathbf{y}) = 1 \wedge C_p = KZG.Commit(\mathbf{pk}_{kzg}, p_p(x))\}$, be the relation for prediction where \mathcal{C}_{pred} is the prediction circuit as described in C.4. Generate a proof $\pi_{\mathcal{R}}$ for the relation \mathcal{R}_{pred} using a plonk-based CP-SNARK.
 5. Return $y, \pi_P = \{C_p, \pi_{lkp}, \pi_{\mathcal{R}}, \pi_{link}\}$.
- **VerifyPred**($\mathbf{vk}, C_{\mathcal{F}}, \pi_P, y, \mathbf{x}$):

1. (Only once): Verify the equivalence of pre-images of $C_{\mathcal{F}}^T, C_{\mathcal{F}}^P$ using π_{link} and $C_{\mathcal{F}}$.
2. Verify that all paths on the pre-image matrix of $C_{\mathcal{P}}$ belong to the forest, using $\pi_{\text{lkp}}, C_{\mathcal{F}}, C_{\mathcal{P}}$.
3. Verify the correctness of predictions, by validating the correctness of the paths using $\pi_{\mathcal{R}}, C_{\mathcal{P}}, \mathbf{x}$ and y .

Construction 3: Proof of Forest Training \mathcal{P} is given public parameters \mathbf{pp} , a dataset D for which instantiates a streaming oracle $S(D)$, a forest $\mathcal{F} = \{\mathcal{T}_1, \dots, \mathcal{T}_K\}$, commitment randomness $r_D, r_{\mathcal{F}}$, and the seed element. Having that, \mathcal{P} invokes the following protocols:

1. **Prove Bagging:** If $K > 1$, \mathcal{P} takes as public input the table T_r and proves the correct computation of the frequencies table $F \in \mathbb{F}^{nK}$.
 - (a) **Commit:** \mathcal{P} invokes the space-efficient PC scheme to compute and send to \mathcal{V} , $C_{\mathbf{F}_B} \leftarrow \text{PC.Commit}(\mathbf{pk}, S(\mathbf{F}_B))$.
 - (b) **Prove:** \mathcal{P} receives a random point $s \in \mathbb{F}$ from \mathcal{V} and generates the proofs π_1, π_2 in the following way:
 - i. Let $\mathcal{C}_{B,1}$ be an arithmetic circuit that takes as input \mathbf{g} and checks if $\mathbf{g}_i = (A\mathbf{g}_{i-1} + B) \bmod |T_r|$. Invoke SPARROW to prove that $\mathcal{C}_{B,1}(\mathbf{x}, \mathbf{w}) = 1$ where $\mathbf{x} = (A, B, \text{seed})$ and $\mathbf{w} = (\mathbf{w}_1 = \mathbf{g}, \mathbf{w}_2)$, with \mathbf{w}_2 auxiliary data required to validate modulo operation.
 - ii. Let $\mathcal{C}_{B,2}$ be an arithmetic circuit that takes as input \mathbf{g}, \mathbf{F}_B and proves that $a_1 = \prod_{i \in [nK]} (\mathbf{g}_i + s\mathbf{F}_B[i] + s^2) = \prod_{i \in [|T_r|]} (i + sT_r[i] + s^2)^{\sum_{i \in [nK]} I(\mathbf{F}_B[i]=T_r[i])}$. Invoke SPARROW to prove that $\mathcal{C}_{B,2}(\mathbf{x}, (\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3)) = 1$ where $\mathbf{x} = (T_r, s, s^2)$ and $\mathbf{w}_1 = \mathbf{g}, \mathbf{w}_2 = \mathbf{F}_B$ and \mathbf{w}_3 auxiliary data to compute the powers.
2. **Prove the Correct Partitioning of Data:** Given $S(D)$ and \mathcal{F} , prove the correct assignment of every point in D and every tree in \mathcal{F} .
 - (a) **Commit:** Let $T' = \{T'^1, \dots, T'^K\}$ be a set of auxiliary matrixes as defined in Section 4.2. \mathcal{P} invokes the space-efficient PC scheme to compute and send to \mathcal{V} the commitment $C_{T'} \leftarrow \text{PC.Commit}(\mathbf{pp}, S(T'))$.
 - (b) **Prove:** \mathcal{P} Receives a random point $s \in \mathbb{F}$ from \mathcal{V} and generates the proofs π_3, π_4, π_5 in the following way:
 - i. Let $\mathcal{C}_1 = \tilde{\mathcal{C}}_1^1 | \dots | \tilde{\mathcal{C}}_1^K$ be an arithmetic circuit such that $\tilde{\mathcal{C}}_1^i$ takes as input T_i^i and computes $a_i = \prod_{j \in [n]} (s^{p+1} + \sum_{k \in [p]} s^k T_{jk}^i)$. Invoke SPARROW to prove that $\mathbf{a} = \mathcal{C}_1(\mathbf{x}, \mathbf{w})$, where $\mathbf{w} = T'$, $\mathbf{x} = (1, s, \dots, s^{p+1})$, but without committing T' again.
 - ii. Let $\mathcal{C}_2 = \tilde{\mathcal{C}}_2^1 | \dots | \tilde{\mathcal{C}}_2^K$ be an arithmetic circuit such that $\tilde{\mathcal{C}}_2^i$ takes as input the i -th tree T^i and computes $a_i = \prod_{j \in [L]} (s^{p+1} + \sum_{k \in [p]} s^k T_{jk}^i)^{|P_{ij}|}$. Invoke a GKR-based zkSNARK to prove $\mathbf{a} = \mathcal{C}_2(\mathbf{x}, \mathbf{w}_1, \mathbf{w}_2)$ (without committing to \mathbf{w}_1), where $\mathbf{w}_1 = F$, \mathbf{w}_2 information required to compute the powers, and $\mathbf{x} = (1, s, \dots, s^{p+1})$. Observe that we require \mathbf{a} to be equal with (a). This is satisfied by committing to $f_a = (a_1, \dots, a_K)$ at the beginning of step (a).
 - iii. Let $\mathcal{C}_3 = \tilde{\mathcal{C}}_3^1 | \dots | \tilde{\mathcal{C}}_3^K$ be an arithmetic such that $\tilde{\mathcal{C}}_3^i$ validates the correctness of the matrix T^i for the i -th tree as described in Section 4.2. Invoke SPARROW to prove that $\mathcal{C}_3(\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3) = 1$ where $\mathbf{w}_1 = D$, $\mathbf{w}_2 = T'$ and \mathbf{w}_3 , the auxiliary information required for by the inference circuit for every j -th point of D and i -th tree.
3. **Prove Leaf Histograms:** If $K > 1$, \mathcal{P} takes as input $S(F)$. Moreover let $\mathbf{h} = (\mathbf{h}^1, \dots, \mathbf{h}^K)$ be the set addresses such that $\mathbf{h}_{j,k}^i = (j, p_{k,j}, T_k^i[0], p_{k,j}.\text{label})$ for $j \in [d], k \in [n]$. \mathcal{P} can have access to $S(\mathbf{h})$ using $S(D), S(T')$, so does not commit \mathbf{h} .
 - (a) **Commit:** Let $H^L = (H^{L,1}, \dots, H^{L,K})$ be the set of leaf histograms, $\mathbf{H}_r = (H_r^1, \dots, H_r^K)$ (and \mathbf{H}_w resp.) be the read write transcripts as defined in Section 4.2. \mathcal{P} invokes the space-efficient PC scheme to compute and send to \mathcal{V} the commitments $C_{H^L} \leftarrow \text{PC.Commit}(\mathbf{pp}, S(H^L), r_{H^L})$, $C_{\mathbf{H}_r} \leftarrow \text{PC.Commit}(\mathbf{pp}, S(\mathbf{H}_r), r_{\mathbf{H}_r})$, $C_{\mathbf{H}_w} \leftarrow \text{PC.Commit}(\mathbf{pp}, S(\mathbf{H}_w), r_{\mathbf{H}_w})$.
 - (b) **Prove:** \mathcal{P} receives a random point $s \in \mathbb{F}$ and generates the proof π_6, π_7 in the following way:
 - i. Let $\mathcal{C}_4 = \tilde{\mathcal{C}}_4^1 | \dots | \tilde{\mathcal{C}}_4^K$ be an arithmetic circuit such that $\tilde{\mathcal{C}}_4^i$ takes as input $\mathbf{h}^i, \mathbf{H}_w^i, \mathbf{H}_r^i, \mathbf{F}_B^i$ and checks if $\mathbf{h}_j^i = \mathbf{H}_w^i[j][0] = \mathbf{H}_r^i[j][0]$ and $\mathbf{H}_w^i[j][1] = \mathbf{H}_r^i[j][1] + \mathbf{F}_B^i[j]$, $\forall j \in [nd]$. Invoke SPARROW to prove that $\mathcal{C}_4(\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3, \mathbf{w}_4) = 1$ where $\mathbf{w}_1 = \mathbf{h}$, $\mathbf{w}_2 = \mathbf{H}_r$, $\mathbf{w}_3 = \mathbf{H}_w$, $\mathbf{w}_4 = \mathbf{F}_B$.
 - ii. Let $\mathcal{C}_5 = \tilde{\mathcal{C}}_5^1 | \dots | \tilde{\mathcal{C}}_5^K$ be an arithmetic circuit such that $\tilde{\mathcal{C}}_5^i$ takes as input $\mathbf{h}^i, \mathbf{H}_w^i, \mathbf{H}_r^i, H^L$ and returns 1 if $a_1 a_2 = a_3 a_4$ where $a_1 = \prod_{j,k,l,m \in [d],[B],[L],[2]} ((1, s, s^2, s^3), (j, k, l, m)) + s^4 H^{L,i}((j, k, l, m)) + s^5$, $a_2 = \prod_{j \in [nd]} ((1, s, s^2, s^3), \mathbf{H}_r^i[j][0]) + s^4 \mathbf{H}_r^i[j][1] + s^5$, $a_3 = \prod_{i,j,k,l \in [d],[B],[L],[2]} ((1, s, s^2, s^3), (i, j, k, l)) + s^5$

and $a_4 = \prod_{j \in [nd]} ((1, s, s^2, s^3), \mathbf{H}_w^i[j][0]) + s^4 \mathbf{H}_w^i[j][1] + s^5$. Invoke SPARROW in a similar fashion with (a) to prove that $C_5(\mathbf{x}, \mathbf{w} = (\mathbf{h}, \mathbf{H}_r, \mathbf{H}_w, H^L)) = 1$, where $\mathbf{x} = (1, s, s^2, s^3, s^4, s^5)$.

4. **Prove Non-Leaf Histograms:** Given \mathcal{F} and streaming access to the leaf histograms $S(H^L)$, \mathcal{P} proves the correct computation of the histograms of the non-leaf nodes H .
 - (a) **Commit:** Let $H = (H^1, \dots, H^K)$ be the set of non-leaf histograms where H^i corresponds to the non-leaf histograms of the i -th tree. \mathcal{P} commits and sends to \mathcal{V} the commitment $C_H \leftarrow PC.Commit(\mathbf{pp}, S(H), r_H)$.
 - (b) **Prove:** Receive a random point $s \in \mathbb{F}$ from \mathcal{V} and generates π_8 as follows. Let $C_6 = \tilde{C}_6^1 | \dots | \tilde{C}_6^K$ be an arithmetic circuit such that \tilde{C}_6^i takes as input $H^{L,i}, H^i$ and some auxiliary information (e.g., computation transcript as defined in Section 4.2), and outputs 1 if H^i is the correct set of non-leaf histograms for the i -th tree. Invoke SPARROW to prove that $C_6(\mathbf{w}) = 1$, but only committing to the additional auxiliary data and not H^L or H .
5. **Prove Correctness of Splits:** Given \mathcal{F} and streaming access to the histograms H , generate a proof π_9 in the following way:
 - (a) **Prove:** Let $C_7 = \tilde{C}_7^1 | \dots | \tilde{C}_7^K$ be an arithmetic circuit such that \tilde{C}_7^i takes as input H^i , the i -th tree and some auxiliary information aux^i (e.g., bit-representations of gini values for comparisons) and outputs 0 if the split values and features are the same in T_i with the ones computed from H^i . Invoke SPARROW to prove that $C_7(\mathbf{w}) = 1$, where $\mathbf{w}_i = (H^i, T_i, aux^i), i \in [K]$ (accessed in a streaming fashion by invoking $S(H)$, computing on the fly aux^i and using T_i). When using SPARROW we only have to commit to aux .
6. **Prove Forest Well-Formedness:** Given the forest \mathcal{F} , \mathcal{P} shows that it is well-formed by invoking a GKR-based zkSNARK on the concatenation of K arithmetic circuits as described in Section 4.2. We set π_{10} be such a proof.

5 Experimental Evaluation

We implemented and experimentally evaluated SPARROW and our *zkFTP* and in this section we report our findings.

Software. Our implementation takes 9000 lines of C++ code, available in [92]. For the necessary field, elliptic curve, and pairing operations we used the `mc1` library [93] implementation of the BN-SNARK1 curve over a 254-bit prime. For forest training [44], we implemented training and certification algorithms and we generated the datasets for classification using the `make_classification` function of `scikit-learn` library [94]. For the parts of the prover that use the standard GKR-based SNARK and to generate our arithmetic circuits we rely on the publicly available implementation of Virgo [95], but we replaced the field size and the underlying PC commitment with the ones of SPARROW. To instantiate our streaming oracles we used the circuit evaluation functionality of [95] and modified it to compute the evaluations of each layer in a streaming manner. Finally, as mentioned in the introduction, our implementation is also elastic. Following the same design choices with GEMINI, we let the user specify a parameter indicating a threshold instance size. This threshold mainly affects the space-efficient variant of the GKR. Specifically, if the number of gates of one or more consecutive layers lie beyond that threshold, SPARROW proves their correct computation using the time-efficient prover of Libra [10]. Furthermore, increasing that threshold leads to an increase in the working buffer space of the space-efficient sumcheck, resulting in faster proving times (see first experiment of section 5.1). We note that our prover implementation only uses memory storage (not disk). To measure memory usage, we used a shell script that periodically invokes the Unix `free` command to obtain memory usage and we report the maximum observed value.

Hardware. We use a dedicated machine running Linux Ubuntu 20.04.6 LTS with 131GB of RAM and Intel(R) Xeon(R) E-2174G CPU, with 8 cores at 3.80GHz, running in isolation with no other operations. For our experiments, we utilized only a single thread.

5.1 Sparrow Benchmarks

We benchmark SPARROW in three different use cases: (a) *Arbitrary data-parallel circuits* consisting of same-width sub-circuits (i.e., rectangular shape). Such circuits can capture applications such as validating SNARK-friendly algebraic functions [96] or performing batch inference or training of neural networks [8, 7]. We varied the total circuit size between 2^{25} - 2^{30} gates. We set its depth $d = 16$ and we also test SPARROW with our depth-reduction technique for $d = 8$, and $d = 1$. (b) *Multiplication tree* circuits that compute the product of the input wires in a tree-like structure, bottom-up. We vary the number of leaves between 2^{24} - 2^{28} . (c) *Batch SHA256 computations* for variable number of hash inputs 2^9 - 2^{13} . To reduce the circuit size, we implemented the XOR and *inner product* gates optimization of [8] for SPARROW. We also report on the prover performance of our space-efficient sumcheck as a stand-alone tool, for polynomial degree 2^{23} - 2^{29} and also measure the impact of elasticity using variable buffer size 4MB-4GB, (which correspond to threshold instance sizes of 2^{17} - 2^{27}).

Comparison with Prior Works. We compare SPARROW with the state-of-the-art space-efficient zk-SNARK GEMINI [26]. We also benchmark it against the non-space-efficient (“monolithic”) standard GKR-based SNARK [10], using the optimized code of [95]. We replaced the KZG PC with our space-efficient variant that also makes the prover faster. We refer to this instantiation as GKR+KOPIS. For GEMINI, we use its publicly available code [97]. Since we focus on data-parallel circuits, we used the variant without pre-processing, which is roughly $\times 11$ faster than the original one (that works for arbitrary circuits). Finally, we compared our sumcheck as a stand-alone with the “direct” space-efficient sumcheck of [98], and with the standard (non-space-efficient) sumcheck.

Measuring Space Usage. In SPARROW, aside from the working buffer, we also implement the streaming oracles and store the public parameters in memory (which we measure using `free`). In contrast, GEMINI’s implementation allocates space only for the working buffer and generates “dummy” streaming data and public parameters “on the fly”. Because in our work we measure the total prover space and not only the working buffer space, we have to accurately measure the total prover space usage for GEMINI. We do so by calculating the space required to store the computation transcript (i.e., evaluations of intermediate circuit gates) and public parameter size for each different application and configuration. In addition, since the implementations of GEMINI and SPARROW are elastic, we fix the same working buffer space on both schemes supporting a threshold instance size of 2^{20} . That is done for fairness purposes because, configuring the buffer space to the minimum (e.g., $\mathcal{O}(\log |\mathcal{C}|)$ for GEMINI and $\mathcal{O}(\sqrt{|\mathcal{C}|})$ for SPARROW), would make GEMINI’s prover significantly slower—at least $\times 10$ according to our calculation, based on the costs of the $\mathcal{O}(1)$ -space KZG evaluation and commitment algorithms of [26].

(1) Benchmarking our Space-Efficient Sumcheck. Figure 2 (left) shows the prover time for the three different sumchecks and buffer size 4MB. (For our sumcheck, we also need space for the public PC parameters; this is < 32 KB.) First, between the two space-efficient approaches, ours consistently outperforms [18], more so as the polynomial size N increases (e.g., for $N = 2^{29}$ ours takes 404.5 seconds while [18] takes 846.5). This agrees with the asymptotic analysis as our prover takes $\mathcal{O}(N \log \log N)$ while [18] takes $\mathcal{O}(N \log N)$. Indeed, our prover’s time increases virtually in parallel with the non-space efficient, linear-time sumcheck. We stress that the latter is included here only for benchmarking purposes as it would *not be possible to run it* for less than “linear” buffer size (roughly 256MB-16GB for varying N). Figure 2 (right) reports the prover time as we vary the buffer space, for fixed polynomial size $N = 2^{27}$. We omit the non-space-efficient sumcheck from this plot as it would need 8GB buffer to run. Hence, we want to show how the performance of the two space-efficient schemes benefits from a larger buffer. We note the difference in behavior between the two schemes. For [18] the prover time continuously decreases each time the buffer is doubled.

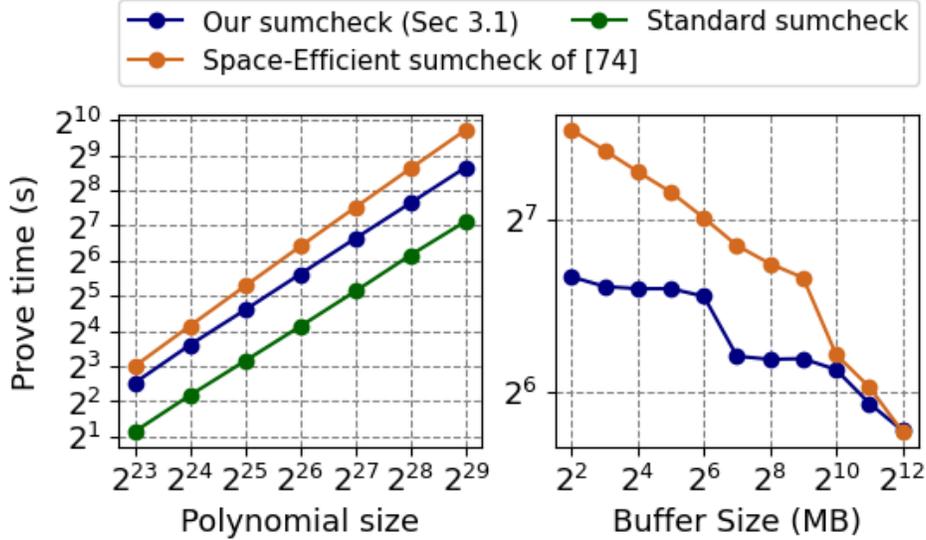


Figure 2: Sumcheck proving time vs. variable polynomial size with buffer 4MB (left), and variable buffer size with polynomial of size 2^{27} (right).

In practice, following our discussion in Section 3.1, its bottleneck is the repeated passes over the data, and each buffer increase results in fewer passes. On the other hand, our prover’s overhead is unaffected until buffer size becomes 2^{20} (from 96 to 73 seconds) as the $\log \log N$ factor decreases from 4 to 3. In practice, ours will be consistently faster, until the buffer size is large enough so that both schemes “degenerate” into the non-space-efficient sumcheck.

As a final takeaway, our space-efficient sumcheck is only $\times 2.6$ slower than the standard non-space-efficient one (Figure 2 (left)), while being able to run with much less space (Figure 2 (right)).

(2) Benchmarking Sparrow. Figure 3 shows the proving time (left) and space (right) of SPARROW, GEMINI, and the standard GKR+KOPIS for the three above applications, as computation size grows. First, we observe that SPARROW significantly outperforms GEMINI both in terms of prover time and space. For arbitrary data-parallel circuits (top row), SPARROW is 9.4 - $11.3\times$ faster and takes 3.2 - $28.7\times$ less space. E.g., for circuit size 2^{30} , SPARROW takes 79 minutes and 2.7GB space (vs. 744 minutes and 80GB for GEMINI). On the other hand, SPARROW is roughly 2.2 - $2.9\times$ slower than the standard GKR+KOPIS (which is “inherited” from the difference in the performance of sumcheck we reported above)—however, it takes far less space. E.g., for circuit size 2^{27} SPARROW needs 1GB vs. 119.5GB for the standard GKR+KOPIS e could not run the latter for circuits size $> 2^{27}$ as we ran out of memory.

The comparison trends for the other two applications are generally similar, modulo some interesting observations. For batch SHA256, the improvement of SPARROW over GEMINI is somewhat smaller (roughly $3\times$ faster prover). This is because SPARROW uses a circuit of 2^{17} gates and seven layers, whereas GEMINI can benefit from the R1CS encoding of SHA that takes roughly 2^{15} gates. This gap is inherent as our scheme works for layered arithmetic circuits, which tend to be larger due to padding. In the future, we plan to extend SPARROW to non-layered data-parallel circuits using [84]. For SHA256, the space usage of SPARROW appears virtually constant. This is an artifact of us having “fixed” the buffer and public parameter space (which in this case is larger than necessary) and the fact that the streamed input per circuit is relatively tiny. For multiplication trees, the “gap” between SPARROW and GKR+KOPIS is smaller than above (1.3 - $1.6\times$ prover slowdown), due to the large input size (compared to the circuit size), thus the commitment-related times become a common

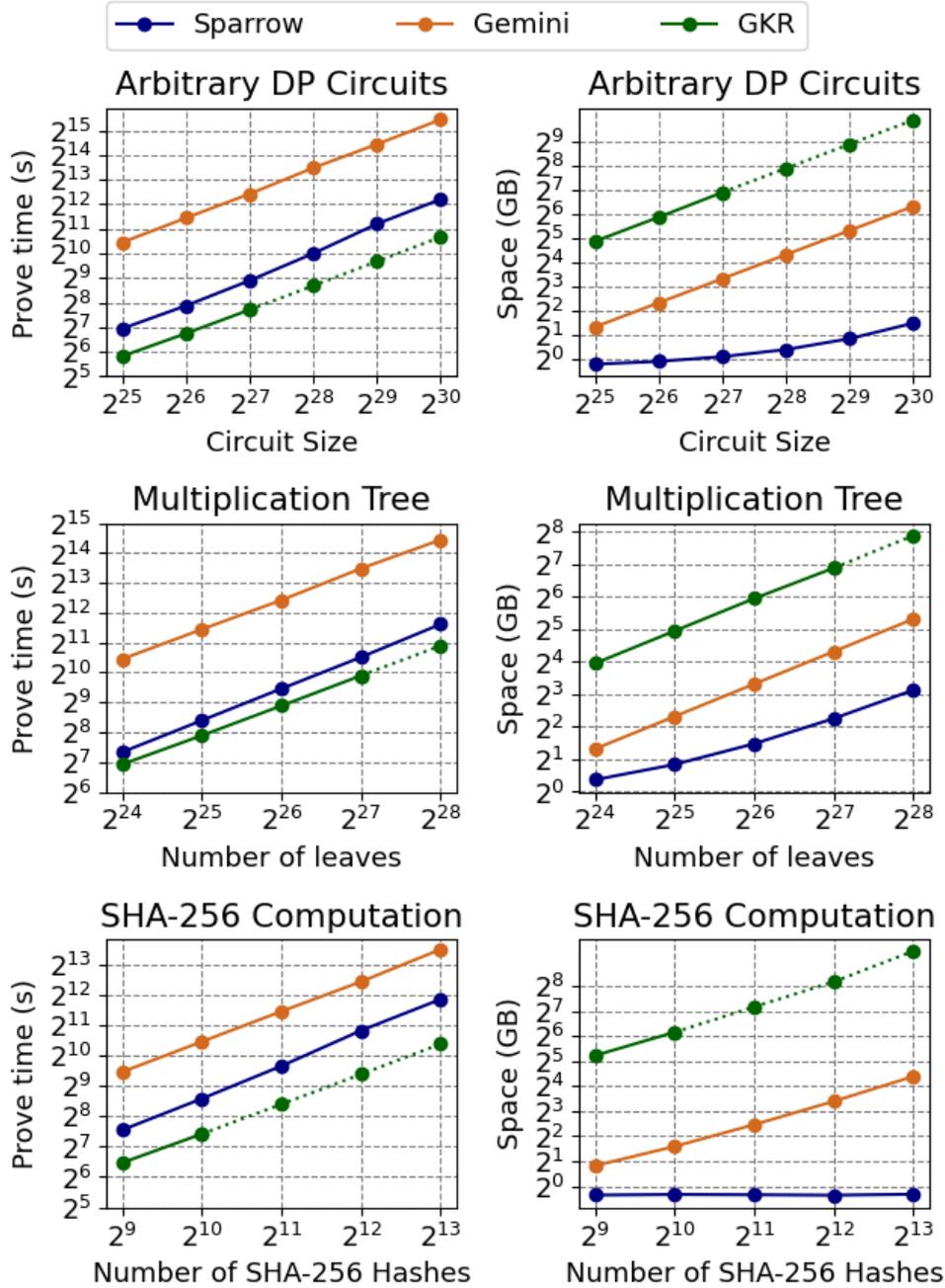


Figure 3: Proving time and space complexity of SPARROW, GEMINI and GKR+KOPIS, for arbitrary data-parallel circuits (top), SHA hashes (middle) and multiplication trees (bottom).

bottleneck for both.

Profiling space usage. For the same instance size threshold, SPARROW requires 128MB of working buffer space, while GEMINI 190MB. That mainly stems from implementation differences and from the fact that GEMINI invokes a batch sumcheck protocol involving more than two polynomials which require their own buffer. This means that if we set GEMINI’s buffer size to be 128MB, its prover would be slower as we would need to make its threshold size roughly two times smaller. Regarding

public parameters, for all our experiments on SPARROW, we fix their size to 200MB, allowing us to support polynomials of size up to 2^{36} . On the other hand, GEMINI uses the KZG PC scheme, and to store its public parameters, it needs 1-48GB. That is the main bottleneck in GEMINI’s space utilization as the public parameters scale linearly to the circuit size. In contrast, the main bottleneck of SPARROW eventually becomes the space needed to instantiate the streaming oracles, which is *the space required to optimally evaluate the circuit*. Interestingly, for arbitrary data-parallel circuits and Multiplication trees, that space eventually occupies 83% and 96% of the total proving space! In other words, for large enough instance sizes, the space required to generate a proof is almost the same as the one needed for evaluation.

Proof size & verification. Although our main focus is on prover time and space, we also report on the proof size and verification time of SPARROW. For arbitrary data-parallel circuits with depth $d = 16$, our scheme produces proofs of size 72-78KB as the circuit size ranges from $2^{25} - 2^{30}$. For the other two applications, the proof sizes range from 62-94KB for multiplication trees and 42-58KB for SHA256. Verification times are also exceptionally low: the *largest* observed verification overhead across all experiments was 15ms. This shows that SPARROW yields practically small proofs and very fast verification. It also demonstrates that, as with standard GKR+KOPIS, the proof size grows only logarithmically with the proof size (for fixed depth). Overall, compared to GKR+KOPIS, SPARROW has almost identical verification time and $< 1.7\times$ larger proof due to our version of sumcheck. Finally, GEMINI achieves shorter proofs (in practice, an order of magnitude) which follows directly from the fact its proof is independent of the depth d , whereas SPARROW follows the GKR methodology. However, after applying our depth-reduction technique (see below) the gap between them is just $\approx 2\times$. While [26] does not measure verification times, we expect GEMINI will also be somewhat faster as it relies on a univariate PC.

Impact of depth reduction. We also evaluated the effect of our depth reduction technique from Section 3.2. Using the circuit of the first experiment on arbitrary circuits with $d = 16$, we applied depth reduction and tested SPARROW and GKR+KOPIS on “flattened” versions of the circuit with $d = 8$ and $d = 1$. Asymptotically, $d = 1$ achieves the best prover time but, in practice, it is roughly $2\times$ slower (e.g., to prove a circuit of size 2^{30} it takes 2.49 hours while for $d = 8, 16$ it takes 1.17 and 1.31 hours respectively). This is because by increasing the input size of the circuit, we must commit and evaluate polynomials of larger sizes, which eventually becomes the main bottleneck of the computation. Interestingly, in this case, SPARROW’s prover time becomes almost the same as GKR+KOPIS (on the same depth) while using much less space! What we lose however in proving time, we gain in proof size as it is $\approx \times 1.7$ smaller compared to circuits of depth 8 and 16 (e.g., for size 2^{30} it generates proofs of size of 42.76KB versus 73KB and 75.8KB).

5.2 Performance of *zkFTP*

Next, we report on the performance of our *zkFTP*. For benchmarking we again use the “space-inefficient” version of our prover with GKR+KOPIS. For training parameters, we follow the standard of [78]. We set the max bin size to $B = 128$, maximum tree height for forests to $h = 5$, and vary the number of points n , features d , and trees K .

(1) Single-tree training. First, we focus on training a single tree for variable n between 2^{16} - 2^{22} for fixed $d = 16$ features (Table 5.2 (top)), and variable d between 8-64 for fixed points $n = 2^{20}$. (Table 5.2 (bottom)), for SPARROW and GKR+KOPIS. The immediate observation is that SPARROW *reduces space usage by up to $240\times$ as the instances grow!* At the same time, it has effectively the same proof size and verification time, while exhibiting a prover slow-down by just 1.1-1.3 \times . We also observe that the prover time does not increase strictly linearly with n , as one might expect. That is because, for fewer points, the proving time is dominated by proving node splits (e.g., for $n = 2^{16}$,

Points	Prove (min)		Space (GB)		Verify (sec)		π (KB)	
$n=2^{16}$	1	0.91	0.37	5.9	0.17	0.17	284	282
$n=2^{18}$	2.5	2.25	0.39	23.1	0.18	0.18	340	333
$n=2^{20}$	10.1	7.7	0.42	90	0.19	0.18	389	374
$n=2^{22}$	44.1	x	0.95	x	0.20	x	449	x

Features	Prove (min)		Space (GB)		Verify (sec)		π (KB)	
$d=8$	6.7	5.3	0.32	47.5	0.18	0.18	366	351
$d=16$	10.1	7.7	0.42	90	0.19	0.18	389	374
$d=32$	17	13	0.5	120	0.19	0.18	437	406
$d=64$	32	x	0.96	x	0.20	x	484	x

Table 2: Performance of our *zkFTP* (\square cells) and GKR+KOPIS (\blacksquare cells) for training a decision tree with variable points n and $d = 16$ features (top), and variable d and $n = 2^{20}$ (bottom). “x” indicates the experiment failed due to memory exhaustion.

this accounts for roughly 40% of the total time). As n increases, this is no longer the main overhead as time scales with the number of points (e.g., we see a $4\times$ increase from $n = 2^{20}$ to $n = 2^{22}$, as expected). Similar behavior is seen as d grows as parts of the certification algorithm (e.g., proving the correctness of assignments) depend on the maximum depth of the tree and not on the features themselves. We do not report experiments for prediction as our scheme directly uses [84] with the optimized lookup of [40], so we inherit the performance of these works. For instance, [84] achieves prediction prover time of less than 1sec for $h = 5$. Clearly, proving predictions is a much “lighter” task.

Points	Prove (min)		Space (GB)		Verify (sec)		π (KB)	
$n = 2^{14}$	25.2	15.9	0.4	51.5	0.21	0.21	607	584
$n = 2^{16}$	64.3	40.8	0.43	124.1	0.25	0.24	686	650
$n = 2^{18}$	221.7	x	0.5	x	0.28	x	790	x
$n = 2^{20}$	873	x	0.6	x	0.3	x	903	x

Trees	Prove (min)		Space (GB)		Verify (sec)		π (KB)	
$K=16$	52.7	35.8	0.5	107.6	0.25	0.23	631	612
$K=32$	108.6	x	0.5	x	0.26	x	735	x
$K=64$	221.7	x	0.5	x	0.28	x	790	x
$K=128$	454.7	x	0.5	x	0.29	x	888	x

Table 3: Performance of our *zkFTP* (\square cells) and GKR+KOPIS (\blacksquare cells) for training a random forest with variable points n and $K = 64$ trees (top), and variable K and $n = 2^{18}$ (bottom). “x” indicates memory exhaustion. In both experiments we set $d = 16$ to be the number of features.

(2) Random Forest training. Table 3 shows the proving time, verification time, proof size, and space for generating proofs of training for random forests with $d = 16$ features. We first (top) fix the number of trees to $K = 64$ and vary n , and then (bottom) we fix points to $n = 2^{18}$ and vary K . Overall, SPARROW again shows tremendous improvement over GKR+KOPIS in prover space (up to $288\times$)—also note that we were not even able to run GKR+KOPIS for several instance sizes on our machine with 131GB RAM. An interesting point is that, although the prover time scales linearly with K (as expected, since the certification circuit for a forest consists of K parallel copies of that for a single tree), the space usage is unaffected! This follows from two observations. First, the same buffer space (which, recall, is fixed in our experiments) is used when proving each tree in the forest. Second, all trees in the forest operate on a common dataset, the size of which is fixed. Finally, we note that when proving for a random forest, enforcing the correctness of bagging takes a

significant part of the prover time (between 20% and 38%), as it entails modular operations that need range proofs over large domains. Since in our construction proving bagging depends only on the random seed and d but not on the dataset itself, it can be proven separately in an “offline” pre-processing phase. This optimization (not used in our implementation) would make our proof of training significantly faster.

(3) Space usage vs. plaintext training. In our configuration, SPARROW requires a fixed ≈ 328 MB for buffer space and public parameters, independently of the dataset. The space-efficient property of SPARROW means that eventually *for large enough dataset sizes, its space usage becomes almost identical to that of simply certifying the tree or forest locally!* For instance, for $K = 1$, $n = 2^{22}$, $d = 16$, the size of the quantized dataset is 400MB and the space required for our certification algorithm was measured as 676MB. At the same time, our prover needs roughly 950MB, which includes the space required to certify the computation locally and the space SPARROW utilized to generate the proof—proving the computation takes only $1.4\times$ the space needed to compute it directly!

6 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.

References

- [1] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *ACM STOC*, pages 723–732, 1992.
- [2] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE SP*, pages 238–252, 2013.
- [3] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, pages 305–326, 2016.
- [4] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE SP*, pages 459–474, 2014.
- [5] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In *ACM CCS*, pages 3003–3017, 2022.
- [6] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. *IACR Cryptol. ePrint Arch.*, 2023.
- [7] Kasra Abbaszadeh, Christodoulos Pappas, Dimitrios Papadopoulos, and Jonathan Katz. Zero-knowledge proofs of training for deep neural networks. *IACR Cryptol. ePrint Arch.*
- [8] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkenn: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *CCS*, pages 2968–2985. ACM, 2021.
- [9] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

- [10] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *CRYPTO*, pages 733–764, 2019.
- [11] Srinath T. V. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *CRYPTO*, pages 704–737, 2020.
- [12] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *CRYPTO*, pages 299–328, 2022.
- [13] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *EUROCRYPT*, pages 499–530, 2023.
- [14] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 859–876. IEEE, 2020.
- [15] Jens Ernstberger, Stefanos Chaliasos, George Kadianakis, Sebastian Steinhorst, Philipp Jovanovic, Arthur Gervais, Benjamin Livshits, and Michele Orrù. zk-bench: A toolset for comparative evaluation and performance benchmarking of snarks. *IACR Cryptol. ePrint Arch.*, 2023.
- [16] Nir Bitansky and Alessandro Chiesa. Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In *CRYPTO*, pages 255–272, 2012.
- [17] Justin Holmgren and Ron Rothblum. Delegating computations with (almost) minimal time and space overhead. In *IEEE FOCS*, pages 124–135, 2018.
- [18] Andrew J Blumberg, Justin Thaler, Victor Vu, and Michael Walfish. Verifiable computation using multiple provers. *Cryptology ePrint Archive*, 2014.
- [19] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Public-coin zero-knowledge arguments with (almost) minimal time and space overheads. In *TCC*, pages 168–197, 2020.
- [20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from DARK compilers. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 677–706. Springer, 2020.
- [21] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Time- and space-efficient arguments from groups of unknown order. In *CRYPTO*, pages 123–152, 2021.
- [22] Muthuramakrishnan Venkatasubramaniam. Ligetron: Lightweight scalable end-to-end zero-knowledge proofs. post-quantum zk-snarks on a browser. In *IEEE SP*, pages 86–86, 2023.
- [23] Laasya Bangalore, Rishabh Bhadauria, Carmit Hazay, and Muthuramakrishnan Venkatasubramaniam. On black-box constructions of time and space efficient sublinear arguments from symmetric-key primitives. In *TCC*, pages 417–446, 2022.
- [24] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*, pages 315–334. IEEE, 2018.

- [25] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkatasubramanian. Ligerio: lightweight sublinear arguments without a trusted setup. *Des. Codes Cryptogr.*, pages 3379–3424, 2023.
- [26] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. Gemini: Elastic snarks for diverse environments. In *EUROCRYPT*, pages 427–457, 2022.
- [27] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT 2010*, pages 177–194, 2010.
- [28] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vsql: Verifying arbitrary SQL queries over dynamic outsourced databases. In *IEEE SP*, pages 863–880, 2017.
- [29] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkatasubramanian, Tiancheng Xie, and Yupeng Zhang. Ligerio++: A new optimized sublinear IOP. In *CCS*, pages 2025–2038. ACM, 2020.
- [30] Changchang Ding and Yan Huang. Dubhe: Succinct zero-knowledge proofs for standard AES and related applications. In *USENIX Sec*, pages 4373–4390, 2023.
- [31] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, pages 859–868, 1992.
- [32] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT*, pages 103–128, 2019.
- [33] Alessandro Chiesa, Elisabetta Fedele, Giacomo Fenzi, and Andrew Zitek-Estrada. A time-space tradeoff for the sumcheck prover. *Cryptology ePrint Archive*, 2024.
- [34] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Cynthia Dwork, editor, *ACM STOC*, pages 113–122, 2008.
- [35] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In *TCC*, pages 1–34, 2021.
- [36] Srinath T. V. Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zkSNARKs. *IACR Cryptol. ePrint Arch.*, 2020.
- [37] Propub. www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing.
- [38] Cortnie Abercrombie. www.techtarget.com/searchenterpriseai/tip/What-is-trustworthy-AI-and-why-is-it-important.
- [39] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero knowledge proofs for decision tree predictions and accuracy. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2039–2053, 2020.
- [40] Matteo Campanelli, Antonio Faonio, Dario Fiore, Tianyu Li, and Helger Lipmaa. Lookup arguments: Improvements, extensions and applications to zero-knowledge decision trees. ePrint, 2023/1518.

- [41] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vcnn: Verifiable convolutional neural network. *IACR Cryptol. ePrint Arch.*, 2020.
- [42] Sanjam Garg, Aarushi Goel, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, Guru-Vamsi Policharla, and Mingyuan Wang. Experimenting with zero-knowledge proofs of training. In *ACM CCS*, pages 1880–1894, 2023.
- [43] Steven Salzberg. Book review: C4.5: programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993. *Mach. Learn.*, pages 235–240, 1994.
- [44] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.
- [45] Bob Hayes. <https://businessoverbroadway.com/2021/02/14/machine-learning-algorithms-and-the-data-pros-who-use-them>.
- [46] Amit Raja Naik. Solving Machine Learning Problems On Kaggle Vs Real Life. <https://analyticsindiamag.com/solving-machine-learning-problems-on-kaggle-vs-real-life>.
- [47] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *STOC*, pages 111–120. ACM, 2013.
- [48] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, pages 276–294, 2014.
- [49] Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *CRYPTO*, pages 359–388, 2022.
- [50] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In *CRYPTO*, pages 681–710, 2021.
- [51] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. *IACR Cryptol. ePrint Arch.*, page 1021, 2019.
- [52] Wilson D. Nguyen, Trisha Datta, Binyi Chen, Nirvan Tyagi, and Dan Boneh. Mangrove: A scalable framework for folding-based snarks. *IACR Cryptol. ePrint Arch.*, page 416, 2024.
- [53] Benedikt Bünz and Jessica Chen. Proofs for deep thought: Accumulation for large memories and deterministic computations. *IACR Cryptol. ePrint Arch.*, page 325, 2024.
- [54] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *EUROCRYPT*, pages 769–793, 2020.
- [55] Alessandro Chiesa, Ziyi Guan, Shahar Samocha, and Eylon Yogev. Security bounds for proof-carrying data from straightline extractors. *IACR Cryptol. ePrint Arch.*, page 1646, 2023.
- [56] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vCNN: Verifiable convolutional neural network based on zk-snarks. *Cryptology ePrint Archive*, 2020.
- [57] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. Zen: An optimizing compiler for verifiable, zero-knowledge neural network inferences. *Cryptology ePrint Archive*, 2021.
- [58] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for Zero-Knowledge proofs with applications to machine learning. In *USENIX Sec. 2021*, pp. 501–518.

- [59] Lingchen Zhao, Qian Wang, Cong Wang, Qi Li, Chao Shen, and Bo Feng. Veriml: Enabling integrity assurances and fair payments for machine learning as a service. *IEEE Transactions on Parallel and Distributed Systems*, pages 2524–2540, 2021.
- [60] Haodi Wang and Thang Hoang. ezdps: An efficient and zero-knowledge machine learning inference pipeline. *arXiv preprint arXiv:2212.05428*, 2022.
- [61] Chenyu Huang, Jianzong Wang, Huangxun Chen, Shijing Si, Zhangcheng Huang, and Jing Xiao. zkmlaas: a verifiable scheme for machine learning as a service. In *IEEE GLOBECOM 2022*, pages 5475–5480.
- [62] Hengrui Jia, Mohammad Yaghini, Christopher A. Choquette-Choo, Natalie Dullerud, Anvith Thudi, Varun Chandrasekaran, and Nicolas Papernot. Proof-of-learning: Definitions and practice. In *IEEE SP*, pages 1039–1056, 2021.
- [63] Congyu Fang, Hengrui Jia, Anvith Thudi, Mohammad Yaghini, Christopher A. Choquette-Choo, Natalie Dullerud, Varun Chandrasekaran, and Nicolas Papernot. On the fundamental limits of formally (dis)proving robustness in proof-of-learning. *CoRR*, abs/2208.03567, 2022.
- [64] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *ACM STOC*, pages 291–304, 1985.
- [65] Uriel Feige, Shafi Goldwasser, László Lovász, Shmuel Safra, and Mario Szegedy. Interactive proofs and the hardness of approximating cliques. *J. ACM*, pages 268–292, 1996.
- [66] Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinuo Zhang, Dawn Song, Xiang Xie, and Yupeng Zhang. Doubly efficient interactive proofs for general arithmetic circuits with linear prover time. In *CCS*, pages 159–177. ACM, 2021.
- [67] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, pages 71–89, 2013.
- [68] Riad S Wahby, Ye Ji, Andrew J Blumberg, Abhi Shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2071–2086, 2017.
- [69] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 90–112, 2012.
- [70] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: nearly practical verifiable computation. *Commun. ACM*, pages 103–112, 2016.
- [71] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019.
- [72] Leo Breiman, J. H. Friedman, Richard A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.
- [73] J. Ross Quinlan. Induction of decision trees. *Mach. Learn.*, pages 81–106, 1986.
- [74] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

- [75] Khaled Alsabti, Sanjay Ranka, and Vineet Singh. CLOUDS: A decision tree classifier for large datasets. In Rakesh Agrawal, Paul E. Stolorz, and Gregory Piatetsky-Shapiro, editors, *KDD*, pages 2–8. AAAI Press, 1998.
- [76] Ruoming Jin and Gagan Agrawal. Communication and memory efficient parallel decision tree construction. In *SIAM International Conference on Data Mining*, pages 119–129. SIAM, 2003.
- [77] Ping Li, Christopher J. C. Burges, and Qiang Wu. Mcrank: Learning to rank using multiple classification and gradient boosting. In *Annual Conference on Neural Information Processing Systems*, pages 897–904, 2007.
- [78] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Annual Conference on Neural Information Processing Systems*, pages 3146–3154, 2017.
- [79] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *EUROCRYPT*, pages 738–768, 2020.
- [80] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of correct computation. In Amit Sahai, editor, *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, volume 7785 of *Lecture Notes in Computer Science*, pages 222–242. Springer, 2013.
- [81] Alessandro Chiesa, Michael A. Forbes, and Nicholas Spooner. A zero knowledge sumcheck and its applications. *CoRR*, 2017.
- [82] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vram: Faster verifiable RAM with program-independent preprocessing. In *SP*, pages 908–925. IEEE, 2018.
- [83] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE SP*, pages 926–943, 2018.
- [84] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *IEEE SP*, pages 859–876, 2020.
- [85] Srinath Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zkSNARKs. *Cryptology ePrint Archive*, 2020.
- [86] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for inner pairing products and applications. In *Advances in Cryptology—ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*, pages 65–97. Springer, 2021.
- [87] Ian Chang, Katerina Sotiraki, Weikeng Chen, Murat Kantarcioglu, and Raluca A. Popa. HOLMES: efficient distribution testing for secure collaborative learning. In *USENIX Sec*, pages 4823–4840, 2023.
- [88] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, pages 225–244, 1994.

- [89] Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Unlocking the lookup singularity with lasso. *IACR Cryptol. ePrint Arch.*, page 1216, 2023.
- [90] Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. Sublonk: Sublinear prover plonk. *IACR Cryptol. ePrint Arch.*, page 902, 2023.
- [91] W. E. Thomson. A modified congruence method of generating pseudo-random numbers. *Comput. J.*, page 83, 1958.
- [92] <https://github.com/anonymousg3bz6q2/sparrow>.
- [93] <https://github.com/herumi/mcl>.
- [94] <https://scikit-learn.org/stable/>.
- [95] <https://github.com/tamucrypto/virgo-plus>.
- [96] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *ASIACRYPT*, pages 191–219, 2016.
- [97] <https://github.com/arkworks-rs/gemini>.
- [98] Andrew J. Blumberg, Justin Thaler, Victor Vu, and Michael Walfish. Verifiable computation using multiple provers. *IACR Cryptol. ePrint Arch.*

A Proofs for our Space-Efficient Sumcheck

A.1 Complexity Analysis

Regarding space complexity, we argue as follows. First, the buffer space required in initialization is $\mathcal{O}(\sqrt{2^n})$ field elements. In the first pass, we perform FFT for polynomials of size n , which requires space $\mathcal{O}(n \log n)$. Similarly, in the second pass, we need to perform FFTs for polynomials of size $2^{n/2 - \log n}$, which requires space $\mathcal{O}(2^{n/2})$, since $(n/2 - \log n)2^{n/2 - \log n} < \frac{n}{2}2^{n/2}/n = 2^{n/2-1} < 2^{n/2}$. For the delegation protocol, we require that the underlying PC scheme has linear space complexity with respect to the degree of the polynomial [80, 10]. Observe that throughout the protocol, we only need to commit and evaluate to polynomials of degree at most $2^{n/2 - \log n + 1}$. Likewise, we argue about the sumchecks we need to prove, resulting in space complexity of $\mathcal{O}(2^{n/2 - \log n + 1})$. The remaining steps use the standard sumcheck for instances of size $\mathcal{O}(\sqrt{2^n})$, leading to overall space of $\mathcal{O}(\sqrt{2^n})$.

Next, we give a detailed analysis of the prover’s complexity. Starting from the **First round** of **Phase 1**, the prover performs $\frac{2^n}{n}$ polynomial multiplications and additions over polynomials of degree n . To efficiently multiply two polynomials, the prover first performs an IFFTs over p_f and p_g , to convert them in the form $p_f(x) = \sum_{i \in [n]} c_i x^i$ (likewise for $p_g(x)$) and an FFT over p_f and p_g to evaluate them in $2n$. To compute their product, the prover performs $2n$ multiplications over these evaluations. Finally it updates $p_0(x)$ by adding the $2n$ evaluations (note that $p_0(x)$ is also represented as a vector of $2n$ evaluations at the same evaluation points with $p_f(x) \cdot p_g(x)$). To compute the FFTs, the product and updated $p_0(x)$, it takes $\mathcal{O}(n \log n)$ field multiplications/additions, $\mathcal{O}(n)$ field multiplications and $\mathcal{O}(n)$ field additions respectively, hence, $\mathcal{O}(n \log n)$ field operations. For computing the IFFTs, the prover needs to perform $\mathcal{O}(n \log n)$ field operations and $\mathcal{O}(1)$ inversions, since it has to compute ω_0^{-1} and n^{-1} . Because these inversions are the same for all IFFTs, we can pre-compute once for all multiplications. Consequently, multiplying two n -degree polynomials takes

$\mathcal{O}(n \log n)$ field operations, so **Phase 1** requires $\mathcal{O}(n \log n \cdot \frac{2^n}{n}) = \mathcal{O}(2^n \log n)$ field operations and $\mathcal{O}(1)$ inversions.

In the first step of **Second round**, the prover starts by computing the vector $\mathbf{y}_0 = (L_1^0(r_0), \dots, L_n^0(r_0))$, which requires $\mathcal{O}(n)$ inversions using the Barycentric formula. Then, using \mathbf{y}_0 , computes $p_f(x), p_g(x)$ which requires $\frac{2^{n/2}}{n}$ inner products of size n totaling to $\mathcal{O}(2^{n/2})$ field operations. Finally, given $p_f(x), p_g(x)$, the prover computes their product in the same way as in the **First round**, taking $\mathcal{O}(\frac{2^{n/2}}{n} \log(\frac{2^{n/2}}{n})) \leq \mathcal{O}(2^{n/2})$ field operations. Since the prover needs to compute, multiply and add $2^{n/2}$ such polynomials, the total complexity of the first step of **Second round** is $\mathcal{O}(2^{n/2} \cdot 2^{n/2}) = \mathcal{O}(2^n)$ field operations and $\mathcal{O}(n)$ inversions. For the second step (the delegation protocol), the prover commits and evaluates polynomials (steps 2.a-2.e) of size $\mathcal{O}(2^{n/2 - \log n + 1})$ using a PC scheme for multi-linear polynomials. By utilizing the KZG scheme for multi-linear polynomials, the prover complexity for committing and generating evaluation proofs is $\mathcal{O}(2^{n/2 - \log n + 1})$ group operations (and in particular Multi-Scalar Multiplications (MSMs)). Furthermore, in steps 2.b, 2.d, 2.e, the prover invokes the standard sumcheck protocol over instances of size $2^{n/2 - \log n + 1}$, requiring $\mathcal{O}(2^{n/2 - \log n + 1})$ field operations, while in step 2.c performs $\mathcal{O}(2^{n/2 - \log n + 1})$ inversions and field operations. Consequently step 2 of **Second round** takes $\mathcal{O}(2^{n/2 - \log n + 1}) \leq \epsilon^{\vee}$ group, field operations and inversions. Thus, **Second round** requires $\mathcal{O}(2^n)$ field operations and $\mathcal{O}(2^{n/2 - \log n + 1})$ MSMs and inversions.

In the first step of **Remaining rounds**, the prover starts by computing the vector $\mathbf{y}_1 = (L_1^1(r_1), \dots, L_{2^{n/2}/n}^1(r_1))$ using $\mathcal{O}(2^{n/2 - \log n})$ field operations and inversions. Then, computes $\mathbf{y} = \mathbf{y}_0 \otimes \mathbf{y}_1$ which requires $\mathcal{O}(2^{n/2})$ field operations. Having \mathbf{y} , it computes \mathbf{F}, \mathbf{G} , with $2^{n/2}$ inner products of size $2^{n/2}$, leading to $\mathcal{O}(2^n)$ field operations. Finally, step 2 of **Remaining rounds** requires the prover to generate a sumcheck proof for instances of size $2^{n/2}$, taking $\mathcal{O}(2^{n/2})$ field operations. Consequently, **Remaining rounds** require $\mathcal{O}(2^n)$ field operations and $\mathcal{O}(2^{n/2 - \log n})$ field inversions.

What remains is to argue about the complexity of **Phase 2**. In the first step, the prover has to compute $\beta(\mathbf{i}, \mathbf{r}_2), \forall \mathbf{i} \in \{0, 1\}^{n/2}$ which takes $\mathcal{O}(2^{n/2})$ field operations [10]. Next, it computes \mathbf{F}, \mathbf{G} with $\mathcal{O}(2^{n/2})$ inner products of size $\mathcal{O}(2^{n/2})$ each, totalling to $\mathcal{O}(2^n)$ field operations. In step 2, the prover invokes the sumcheck protocol over instances of size $\mathcal{O}(2^{n/2})$, which requires $\mathcal{O}(2^{n/2})$ field operations. Finally, similarly to step 2 of **Second phase**, step 3 is dominated by $\mathcal{O}(2^{n/2})$ inversions and MSMs. So, **Phase 2** requires $\mathcal{O}(2^n)$ field operations and $\mathcal{O}(2^{n/2})$ MSMs and inversions.

Observe that the main bottleneck of the prover is in **First round** of **Phase 1**, which requires $\mathcal{O}(2^n \log n)$ field operations. All other steps require $\mathcal{O}(2^n)$ field operations. Other than that, the prover performs $\mathcal{O}(2^{n/2})$ MSMs and inversions—sub-linear to the size of the polynomials and thus can be considered negligible compared to the number of field operations. Hence, the total prover's complexity is $\mathcal{O}(2^n \log n)$ field operations.

As for proof size, at the first round, the prover needs to send a polynomial of size $\mathcal{O}(n)$, while the delegation protocol of the second round requires the prover to invoke sumcheck three instances and two evaluation proofs which have size $\mathcal{O}(n)$. Finally, the prover invokes three sumcheck instances resulting in an overall proof size of $\mathcal{O}(n)$. We make a similar argument for verification time.

A.2 Completeness

Let $f, g : \mathbb{F}^n \rightarrow \mathbb{F}$, be the multi-linear polynomials such that $K = \sum_{x \in \{0, 1\}^n} f(x)g(x)$, and $\hat{f}, \hat{g} : \mathbb{F}^{2+n/2} \rightarrow \mathbb{F}$, their equivalent multi-variate polynomials. Because for each $\mathbf{x} \in \mathbb{H}_0 \times \mathbb{H}_1 \times \{0, 1\}^{n/2}$, exists one $\mathbf{y} \in \{0, 1\}^n$, such that $\hat{f}(\mathbf{x}) = f(\mathbf{y})$ and $\hat{g}(\mathbf{x}) = g(\mathbf{y})$, we have that $K = \sum_{\mathbf{x} \in \mathbb{H}_0 \times \mathbb{H}_1 \times \{0, 1\}^{n/2}} \hat{f}(\mathbf{x})\hat{g}(\mathbf{x})$. Next, let $p_0 : \mathbb{F} \rightarrow \mathbb{F}$ be the n -degree polynomial sent in the first round. Because $p_1(z) = \sum_{(x_2, \dots, x_{2+n/2})} \hat{f}(z, x_2, \dots, x_{2+n/2})\hat{g}(z, x_2, \dots, x_{2+n/2})$, then $K = \sum_{\omega \in \mathbb{H}_0} p_1(\omega)$. We follow a similar argument for $p_1(z)$. Note that the completeness of the delegation protocol comes from

the completeness of the standard sumcheck and PC scheme. Likewise we state, for the remaining rounds of **Phase 1** and **2**.

A.3 Soundness

First we will prove soundness for the (non-succinct) information theoretic protocol, where the prover does not invoke any delegation protocol (e.g., Step 2 of **Second round, Phase 1** and step 3 of **Phase 2**). Having established that, let $f, g : \mathbb{F}^n \rightarrow \mathbb{F}$ be multi-linear polynomials, $\hat{f}, \hat{g} : \mathbb{F}^{2+n/2} \rightarrow \mathbb{F}$ the equivalent multi-variate polynomials and $K \in \mathbb{F}$. We will prove that if $K \neq \sum_{x \in \{0,1\}^n} f(x)g(x)$, then the verifier accepts the proof only with negligible probability for any unbounded adversary. Because \hat{f}, \hat{g} are the equivalent polynomials of f, g , it will hold that:

$$\sum_{x \in \{0,1\}^n} f(x)g(x) = \sum_{x_0, x_1, \mathbf{x}_2} \hat{f}(x_0, x_1, \mathbf{x}_2) \hat{g}(x_0, x_1, \mathbf{x}_2)$$

With that in mind, we will follow a non-inductive soundness proof. Specifically, in the first round of phase 1, the adversary wins if it sends a $n+1$ -degree polynomial $p_0(x)$ (s.t $K = \sum_{z \in \mathbb{H}_0} p_0(z)$), that is not equal to $p'_0(x) = \sum_{\mathbf{x} \in \mathbb{H}_1 \times \{0,1\}^{n/2}} \hat{f}(x, \mathbf{x}) \hat{g}(x, \mathbf{x})$ but $p'_0(r_0) = p_0(r_0)$, where $\mathbf{x} = (x_2, \dots, x_{n/2+2})$. This happens however with probability $\frac{n+1}{|\mathbb{F}|}$ due to Schwartz-Zippel lemma. Similarly in the second round of phase 1, the adversary wins it sends a $2^{n/2-\log n+1}$ -degree polynomial $p_1(x)$ s.t $p_1(x) \neq \sum_{\mathbf{y} \in \{0,1\}^{n/2}} \hat{f}(r_1, x, \mathbf{y}) \hat{g}(r_1, x, \mathbf{y})$ but $p'_1(r_1) = p_1(r_1)$, where $\mathbf{y} = (x_3, \dots, x_{n/2+2})$. Due to the Schwartz-Zippel lemma this happens with probability $\frac{2^{n/2-\log n+1}}{|\mathbb{F}|}$. For the remaining rounds of phase 1, the adversary must send a 2-degree polynomial $p_i(x)$, s.t $p_i(x) \neq \sum_{\mathbf{x} \in \{0,1\}^{n/2-i+2}} \hat{f}(\mathbf{r}, x, \mathbf{x}) \hat{g}(\mathbf{r}, x, \mathbf{x})$ but $p'_i(r_i) = p_i(r_i)$, where $\mathbf{x} = (x_i, \dots, x_{n/2+2})$. The latter happens only with $\frac{2}{|\mathbb{F}|}$ probability.

At the end of phase 1, the verifier holds the evaluations $\tilde{y}_{\hat{f}}, \tilde{y}_{\hat{g}}$, derived by the last round of the sumcheck protocol. Observe that the events $\tilde{y}_{\hat{g}} = y_{\hat{g}}$ and $\tilde{y}_{\hat{f}} = y_{\hat{f}}$ happen only with probability $\frac{2n+2^{n/2-\log n}}{|\mathbb{F}|}$ due to Schwartz-Zippel lemma. In any other case, the adversary must convince the verifier for the sumcheck instances $\tilde{y}_{\hat{f}} \neq \sum_{x \in \{0,1\}^{n/2}} f_y(x) f(x, \mathbf{r}_2)$ or $\tilde{y}_{\hat{g}} \neq \sum_{x \in \{0,1\}^{n/2}} f_y(x) g(x, \mathbf{r}_2)$, which happens with probability $\frac{n}{2|\mathbb{F}|}$, due to the soundness of the standard sumcheck protocol.

By taking the union bound over all probabilities, we conclude that the probability convincing the verifier is at most $\frac{n2^{n/2-\log n}}{|\mathbb{F}|}$.

Proof Sketch for Knowledge Soundness. Because we wish to achieve succinctness, the verifier of our final sumcheck protocol delegates some of its computations of the information theoretic protocol to the prover. Because we use sumcheck-based SNARKs to prove these computations, we need to prove that the **Protocol 1** is knowledge sound.

To achieve that we to prove that for any PPT adversary \mathcal{A}_{sc} , there exists an extractor \mathcal{E}_{sc} which given an accepting proof transcript, extracts the polynomials f_y, f_c , such that $\sum_{\omega \in \mathbb{H}_1} p_1(\omega) = K_0, p_1(r_1) = K_1$ and $f_y(\mathbf{r}_3) = y_{\mathbf{Y}}$. We build our extractor as follows. First creates adversaries \mathcal{A}_{PC} against the extractability game of the PC scheme. Specifically it gives them the public parameters \mathbf{pp} , they invoke \mathcal{A}_{sc} , and return the corresponding polynomial commitments along with accepting evaluation proofs. Then uses the extractors \mathcal{E}_{PC} to extract the valid f_c, f_y with overwhelming probability and parses f_c as $p_i(x) = \sum_{i \in [2^{n/2-\log n}]} c_i x^i$. What remains to show that \mathcal{E}_{sc} fails only with negligible probability. This comes directly from the soundness of our sumcheck protocol.

A.4 Zero-knowledge

Protocol 2: Let $f, g : \mathbb{F}^n \rightarrow \mathbb{F}$ be the multi-linear extensions of $\mathbf{A}_f, \mathbf{A}_g$ and $S(\mathbf{A}_f), S(\mathbf{A}_g)$ their streaming oracles. We want to prove that $K = \sum_{x \in \{0,1\}^n} f(x)g(x)$. Our protocol takes as input $S(\mathbf{A}_f), S(\mathbf{A}_g)$, n and public parameters of a PC scheme.

- **Initialization:** Set $\tilde{\mathbf{A}}_f \leftarrow \{0\}^{2^{n/2}}, \tilde{\mathbf{A}}_g \leftarrow \{0\}^{2^{n/2}}, p_0(x) \leftarrow 0, p_1(x) \leftarrow 0, \mathbf{F} \leftarrow \{0\}^{2^{n/2}}, \mathbf{G} \leftarrow \{0\}^{2^{n/2}}$.
- **Commit:** \mathcal{P} : Sample random polynomials $h_1, h_2 : \mathbb{F}^{2^{n/2}} \rightarrow \mathbb{F}$, $R_f(\omega, r_{n/2+2}), R_g(\omega, r_{n/2+2})$ commit them using zkPC scheme and send $C_{h_1}, C_{h_2}, C_{R_f}, C_{R_g}$ and $K_h = \sum h_1(x)h_2(x)$ to \mathcal{V} . Next, \mathcal{V} sends a random point $r \in \mathbb{F}$ at \mathcal{P} , and sets $K_0 = K + rK_h$.
- **Phase 1:** For \hat{f}, \hat{g} the equivalent polynomials of f, g , and \hat{f}', \hat{g}' their masks such that $\hat{f}'(x_0, x_1, \mathbf{x}_2) = \hat{f}(x_0, x_1, \mathbf{x}_2) + z(x_0, x_1, \mathbf{x}_2) \sum_{\omega \in \{0,1\}} R_f(\omega, x_{n/2+1})$, and $\hat{g}'(x_0, x_1, \mathbf{x}_2) = \hat{g}(x_0, x_1, \mathbf{x}_2) + z(x_0, x_1, \mathbf{x}_2) \cdot \sum_{\omega \in \{0,1\}} R_g(\omega, x_{n/2+1})$, prove that $K = \sum_{x \in \mathbb{H}_0 \times \mathbb{H}_1 \times \{0,1\}^{n/2}} \hat{f}'(x)\hat{g}'(x)$
 - **First round** (over the n -degree variable x_0):
 1. \mathcal{P} : For $i = 1, \dots, 2^{n-\log n}$: Read the next n elements of $S(\mathbf{A}_f), S(\mathbf{A}_g)$ and store them in $\tilde{\mathbf{A}}_f, \tilde{\mathbf{A}}_g$. Set $p_f(x) = \sum_{j \in [n]} L_j^{(0)}(x) \tilde{\mathbf{A}}_f[j]$ (and likewise $p_g(x)$ using $\tilde{\mathbf{A}}_g$) and compute $p_0(x) \leftarrow p_0(x) + p_f(x)p_g(x) + rh_1(x_1, \mathbf{i})h_2(x_1, \mathbf{i})$, where $\mathbf{i} \in [2^{n/2-\log n}] \times \{0,1\}^{n/2}$.
 2. \mathcal{V} : Receive $p_0(x)$, check if $K = \sum_{i \in [2n]} p_0(x_i^i)$, pick a random point r_0 , compute $p_0(r_0)$ and send r_0 to \mathcal{P} .
 - **Second round** (over the $2^{n-\log n}$ -degree variable x_1):
 1. \mathcal{P} : For $i = 1, \dots, 2^{n/2}$: Read the next $2^{n/2}$ elements of $S(\mathbf{A}_f), S(\mathbf{A}_g)$, store them in $\tilde{\mathbf{A}}_f, \tilde{\mathbf{A}}_g$ and compute the polynomial $p_f(x) = \hat{f}(r_0, x_1, \mathbf{i})$ defined as $\sum_{j \in [2^{n-\log n}]} L_j^1(x)c_j$, where c_j equals to $\sum_{k \in [n]} \mathbf{y}_0[k] \tilde{\mathbf{A}}_f[jn+k]$ and $\mathbf{y}_0 = (L_1^0(r_0), \dots, L_n^0(r_0))$. Likewise compute $p_g(x)$ using $\tilde{\mathbf{A}}_g$. Finally compute $p_1(x) \leftarrow p_1(x) + p_f(x)p_g(x) + rh_1(r_1, x_2, \mathbf{i})h_2(r_1, x_2, \mathbf{i})$ where $\mathbf{i} \in \{0,1\}^{n/2}$.
 2. \mathcal{V} : Delegates its checks to \mathcal{P} following the same protocol as non-zero-knowledge version.
 - **Remaining rounds** (over the 1-degree variables $x_2, \dots, x_{n/2+2}$):
 1. \mathcal{P} : For $i = 1, \dots, 2^{n/2}$: Read the next $2^{n/2}$ elements of $S(\mathbf{A}_f), S(\mathbf{A}_g)$, store them in $\tilde{\mathbf{A}}_f, \tilde{\mathbf{A}}_g$ and compute $\mathbf{F}[i] = \hat{f}(r_0, r_1, \mathbf{i}) = \sum_{j \in [2^{n/2}]} (\mathbf{y}_0 \otimes \mathbf{y}_1)[j] \tilde{\mathbf{A}}_f[j]$ (and similarly compute $\mathbf{G}[i]$ using $\tilde{\mathbf{A}}_g$), where $\mathbf{y}_1 = (L_1^1(r_1), \dots, L_{2^i}^1(r_1))$.
 2. \mathcal{P} : interacts with \mathcal{V} following the zero-knowledge version of the sumcheck protocol using as mask the multi-linear polynomials $h_1(r_1, r_2, x), h_2(r_1, r_2, x)$ to prove that $p_1(r_1)$ is equal to $\sum_{x \in \{0,1\}^{n/2}} \hat{f}'(r_0, r_1, x)\hat{g}'(r_0, r_1, x)$. Finally, \mathcal{V} ends up with claimed evaluations $y_{\hat{f}} = y_{\hat{f}} + z(r_0, r_1, \mathbf{r}_2) \sum_{\omega} R_f(\omega, r_{n/2+1}), y_{\hat{g}'} = y_{\hat{g}'} + z(r_0, r_1, \mathbf{r}_2) \sum_{\omega} R_g(\omega, r_{n/2+1})$ at (r_0, r_1, \mathbf{r}_2) . Furthermore, \mathcal{P} generates an evaluation proof of h_1, h_2 at the same point. Using these points, \mathcal{V} validates the last round of the zero-knowledge sumcheck.
- **Phase 2:** Reduce the evaluation claims of \hat{f}, \hat{g} into claims of f, g .
 1. \mathcal{P} : For $i = 1, \dots, 2^{n/2}$: Read $2^{n/2}$ elements from $S(\mathbf{A}_f), S(\mathbf{A}_g)$ and store them in $\tilde{\mathbf{A}}_f, \tilde{\mathbf{A}}_g$. Update \mathbf{F} (and similarly \mathbf{G}) by computing $\mathbf{F}[j] = \mathbf{F}[j] + \beta(\mathbf{i}, \mathbf{r}_2) \mathbf{A}_f[j]$ for each $j \in [2^{n/2}]$. At the end of the iteration, \mathcal{P} holds $f(x, \mathbf{r}_2), g(x, \mathbf{r}_2)$.
 2. \mathcal{P} receives $a_1, a_2 \in \mathbb{F}$ from \mathcal{V} and interacts with it following the zero-knowledge sumcheck protocol for the instance $a_1 y_{\hat{f}} + a_2 y_{\hat{g}'} = \sum_{x \in \{0,1\}^{n/2}, \omega \in \{0,1\}} \left(f_{\mathbf{y}}(x) (a_1 f(x, \mathbf{r}_2) + a_2 g(x, \mathbf{r}_2)) + z(r_0, r_1, \mathbf{r}_2) (a_1 R_f(\omega, r_{n/2+1}) + a_2 R_g(\omega, r_{n/2+1})) \right)$.
 3. \mathcal{P} sends $y_{\mathbf{y}_0}, y_{\mathbf{y}_1}, y_f, y_g$, generates an evaluation proof for $R_g(r_{\omega}, r_{n/2+1}), R_f(r_{\omega}, r_{n/2+1})$ and \mathcal{V} validates the last round of the sumcheck protocol.
 4. \mathcal{V} checks the correctness of $y_{\mathbf{y}_0}$ locally and for $y_{\mathbf{y}_1}$, interacts with \mathcal{P} following steps 2.c-2.e (but in step 2.d replaces $p_1(x)$ with the multi-linear polynomial s.t $l(x) = \beta(x, r_{\mathbf{y}_1})$).

At a high level, to make our succinct scheme zero-knowledge, we rely on ideas from [10, 81]. I.e., we use randomly sampled polynomials h_1, h_2 to “blind” the polynomials the prover sends and generate a batched sumcheck proof. Following the optimizations of [10], it suffices to represent h_1, h_2 as $h(x_1, \dots, x_{n/2+2}) = c_1 + h^{(1)}(x_1) + \dots + h^{(n/2+2)}(x_{n/2+2})$, where $h^{(1)}(x_1)$ is a n -degree polynomial and $h^{(2)}(x_2)$ a $2^{n/2 - \log n}$ -degree polynomial, while the remaining partial polynomials have degree 1. Hence, h_1, h_2 add only a negligible overhead to the proving time and, most importantly for our space-efficient construction, they do not violate our space requirements. Next, to avoid sending evaluations of the polynomials \hat{f}, \hat{g} , which would leak additional information about f, g , we mask these using the random polynomials R_f, R_g . Since for each polynomial the verifier receives only one evaluation, we can heavily reduce the size of R_f, R_g to two variables of degree 1 (e.g., $R_f(x_1, \omega) = a_1 + a_2x_1 + a_3\omega + a_4x_1\omega$), again minimizing the overhead to the proving time and space.

Protocol 2 gives a detailed description of the zero-knowledge version of our space-efficient sumcheck, and we highlight with blue color the necessary changes. This protocol has an additional **Commit** phase in which the prover first samples the blinding polynomials h_1, h_2 , the masking polynomials R_f, R_g and commits them using the zkPC scheme. Finally computes the sum $K_h = \sum_{\mathbf{x} \in \mathbb{H}_0 \times \mathbb{H}_1 \times \{0,1\}^{n/2}} h_1(\mathbf{x})h_2(\mathbf{x})$ and sends K_h along with the commitments to the verifier. Then receives a random point $r \in \mathbb{F}$ from the verifier and, in the first phase, interacts with it to prove that $K + rK_h = \sum_{\mathbf{x} \in \mathbb{H}_0 \times \mathbb{H}_1 \times \{0,1\}^{n/2}} (\hat{f}'(x)\hat{g}'(x) + rh_1(x)h_2(x))$, where:

$$\hat{f}'(x_0, x_1, \mathbf{x}_2) = \hat{f}(x_0, x_1, \mathbf{x}_2) + z(x_0, x_1, \mathbf{x}_2) \sum_{\omega \in \{0,1\}} R_f(\omega, x_{n/2+1})$$

Recall that the polynomials $h_1(x)$ (and $h_2(x)$ resp.) have the form $h_1(x) = \sum_{i \in [n/2]} h_1^{(i)}(x_i)$, where $h_1^{(1)}$ has degree n , $h_2^{(2)}$ has degree $2^{n/2 - \log n}$, while the rest polynomials have degree 1. Moreover, and for reasons that will be clear later, it is enough for R_f, R_g to be multi-linear polynomials. At the end of **Phase 1**, the prover gives the evaluations of the masked multi-linear polynomials to the verifier along with opening proofs for the evaluations of h_1, h_2 . The verifier validates the last step of the sumcheck and proceeds to **Phase 2** in which it interacts with the prover over the zero-knowledge version of the sumcheck protocol [10] over the instance:

$$a_1 y_{\hat{f}'} + a_2 y_{\hat{g}'} = \sum_{x \in \{0,1\}^{n/2}, \omega \in \{0,1\}} I(0, \omega) f_{\mathbf{y}}(x) \left(a_1 f_{\mathbf{F}}(x, \mathbf{r}_2) + a_2 g(x, \mathbf{r}_2) \right) + I((x, 0)) z(r_0, r_1, \mathbf{r}_2) \left(a_1 R_f(\omega, r_{n/2+1}) + a_2 R_g(\omega, r_{n/2+1}) \right)$$

Where $I(\cdot)$ is the identity function as defined in [10] (Section 4.2). At the end of the protocol, the prover sends y_f, y_g along with the evaluations of R_f, R_g , and their evaluation proofs. The verifier uses them to validate the last round of the sumcheck protocol.

Completeness derives from the completeness of our sumcheck protocol and the zkPC scheme, while soundness, from the soundness of the “plaintext” sumcheck (see previous the section) and the random linear combination as proven in [81]. For the rest of this section we will give a proof for the zero-knowledge.

Proof Sketch. To prove zero-knowledge we will create a simulator \mathcal{S} , that given the sum $K = \sum_{x \in \{0,1\}^n} f(x)g(x)$, is able to simulate the view of \mathcal{V} for all the rounds of our protocol. In more detail, \mathcal{S} works as follows:

1. Randomly select the polynomials $h_1^*(x), h_2^*(x), R_f(x_{n/2+1}, \omega), R_g(x_{n/2+1}, \omega)$, commit them using a zkPC scheme and send the commitments and $K_h = \sum_x h_1(x)h_2(x)$ to \mathcal{V} .

2. Receive r from \mathcal{V} .
3. Select the polynomials $f^*, g^* : \mathbb{F}^n \rightarrow \mathbb{F}$ such that $K = \sum_{x \in \{0,1\}^n} f^*(x)g^*(x)$, and follow all steps of **Phase 1**. More precisely:
 - (a) Compute and send the polynomial $p_0(x_1)$ as in **First round** of **Phase 1** and receive r_0 from \mathcal{V} .
 - (b) Compute the polynomial $p_1(x_2)$ as in **Second round** of **Phase 1** and delegate its evaluation as described in **Protocol 1**.
 - (c) For the **remaining rounds** follow step 3 of **Construction 1** of [10].
4. Send $y_{\hat{f}}^*, y_{\hat{g}'}^*, h_1^*(r_0, r_1, \mathbf{r}_2), h_2^*(r_0, r_1, \mathbf{r}_2)$ to \mathcal{V} .
5. Open the evaluations of h_1^*, h_2^* using the zkPC scheme.
6. Receive from \mathcal{V} , a_1, a_2 .
7. Use the simulator of the zero-knowledge sumcheck protocol of [10] (Section 4.1) to prove **Step 2** of **Phase 2**.
8. Samples y_f, y_g such that the final sumcheck round will be true and sends y_f, y_g , and $R_f^*(r_1, c), R_g^*(r_1, c)$ to \mathcal{V} .
9. Opens $R_f^*(r_1, c), R_g^*(r_1, c)$ using the zkPC scheme.

Steps 2, 6 are indistinguishable from the real world. Moreover, because we assume the existence of a zero-knowledge polynomial commitment scheme, and because the polynomials $h_1^*, h_2^*, R_f^*, R_g^*$ uniformly sampled, steps 1, 5, 9 are indistinguishable from the real world. Indistinguishability of steps 3 and 7 derive from Theorem 3 of [10]. What remains is to prove that steps 4 and 8 are indistinguishable from the real world. Because in both worlds, h_1, h_2, h_1^*, h_2^* are sampled uniformly at random, their evaluations will be indistinguishable between the two worlds. Thus we will focus on showing indistinguishability between the evaluations of $R_f^*, R_g^*, \hat{f}'^*, \hat{g}'^*$ and $R_f, R_g, \hat{f}', \hat{g}'$. Focusing on a single polynomial, we observe that we only have two evaluations of R_f , at $\sum_{\omega \in \{0,1\}} R_f(r_{n/2+1}, \omega)$ at the end of the protocol at $R_f(r_{n/2+1}, r_\omega)$. Following the same logic with [10] and assuming that $R_f = a_1 + a_2x_1\omega + a_3x_1 + a_4\omega$, we have two evaluations ($\sum_{\omega \in \{0,1\}} R_f(r_{n/2+1}, \omega)$ and $R_f(r_{n/2+1}, r_\omega)$), which can arrange them into a matrix that has full rank. Because the matrix has full rank and the points a_i are uniformly sampled, the two evaluations are linearly independent. Moreover, since y_f is sampled uniformly at random with respect to a linear constraint and that $R_f^*(r_{n/2+1}, r_\omega)$ independent and uniformly distributed, $y_{\hat{f}}^*, y_f, R_f^*(r_{n/2+1}, r_\omega)$ will be indistinguishable from the real world. We follow the same process for the polynomial \hat{g}'^* .

B Proofs for Sparrow

B.1 Proof of Lemma 1

Data-Parallel Arithmetic Circuits. First, recall that an arithmetic circuit \mathcal{C} , is d -layered, if it consists of d , depth-one circuits C_i , one for each layer $i \in [d]$. Furthermore, C_i takes as input the output of C_{i-1} . \mathcal{C} is data-parallel if every $C_i : \mathbb{F}^{S_i} \rightarrow \mathbb{F}^{S_{i+1}}$, consists of K_i copies of a sub-circuit $C'_i : \mathbb{F}^{S'_i} \rightarrow \mathbb{F}^{out_i}$ (note that $K_i S'_i = S_i$ and $K_i out_i = S_{i+1}$) and each copy works over a different input.

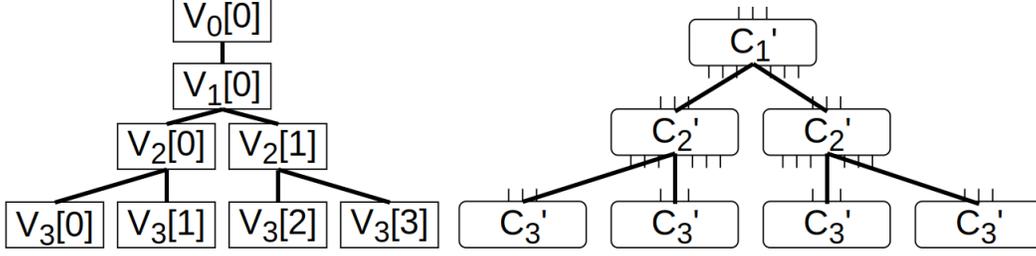


Figure 4: Evaluations tree \mathcal{T}_V (left) and circuit tree \mathcal{T}_C (right) of a data-parallel arithmetic circuit with depth $d = 3$. Note that \mathbf{V}_0 corresponds to the output of the circuit and \mathbf{V}_3 corresponds to the input. Moreover, $\mathbf{V}_i[j]$, for $i \in [3]$, corresponds to the input of the j -th node, in the i -th level of \mathcal{T}_C .

As for the gate evaluations, we define with $\mathbf{V}_i \in \mathbb{F}^{S_i}$, the input of \mathcal{C}_i and with $\mathbf{V}_i[j] \in \mathbb{F}^{S'_i}$, the input of the j -th sub-circuit of \mathcal{C}_i .

We can consider \mathcal{C} as a d -level tree (not necessarily binary) denoted with \mathcal{T}_C , where all K_i nodes of the i -th level correspond to one sub-circuit \mathcal{C}'_i . A child node connects with its parent if the output of the first belongs to the input of the latter. Similarly, we can arrange the evaluations of \mathcal{C} in a tree denoted with \mathcal{T}_V , where the j -th node of the i -th level corresponds to $\mathbf{V}_i[j]$. Figure 4 depicts these trees for a data-parallel circuit of depth d .

Space-Efficient Evaluation of Data-Parallel circuits. A crucial property of data-parallel circuits that will enable us to achieve the desired asymptotics is that we can evaluate them using $|\mathcal{C}|$ field operations and space $|\mathbf{x}| + \sum_{i \in [d]} |\mathcal{C}'_i|$. Such an evaluation algorithm traverses the tree \mathcal{T}_V in post order, computes and stores the next non-leaf node, and deletes its children. Because we compute every node only once, the computing time will be $|\mathcal{C}|$ field operations. Moreover, because we delete all the children of a newly computed node, for every evaluation step, we need to store data of size equal to the input size of all different sub-circuits \mathcal{C}'_i . Thus, by also considering the space required to store the input \mathbf{x} , we get an overall space of $\mathbf{x} + \sum_{i \in [d]} |\mathcal{C}'_i|$.

Main transformation. Having established that, we will show how to convert \mathcal{C} into a data-parallel circuit $\tilde{\mathcal{C}}$ of depth one that takes as input $\tilde{\mathbf{x}}$ consisting of \mathbf{x} , the computation transcript of \mathcal{C} (e.g., all intermediate evaluations) and validates its correctness.

First, we set $\tilde{\mathcal{C}}$ to be the concatenation of nodes (namely sub-circuits) in a post-order traversal of \mathcal{T}_C . For instance, we can transform the circuit of Figure 4 to the 1-layered circuit $\tilde{\mathcal{C}} = (\mathcal{C}'_3 || \mathcal{C}'_3 || \mathcal{C}'_2 || \mathcal{C}'_2 || \mathcal{C}'_1)$. Similarly, $\tilde{\mathbf{x}}$, will be the concatenation of all nodes in \mathcal{T}_V in post-order traversal (e.g., $\tilde{\mathbf{x}} = \mathbf{V}_3[0] || \mathbf{V}_3[1] || \mathbf{V}_2[0] || \mathbf{V}_3[2] || \mathbf{V}_3[3] || \mathbf{V}_2[1] || \mathbf{V}_1[0] || \mathbf{V}_0[0]$). If $\tilde{\mathbf{x}}$ corresponds to a correct computation transcript of \mathcal{C} , then any sub-vector of $\tilde{\mathbf{x}}$ (excluding the sub-vectors of \mathbf{V}_d) will correspond to one sub-vector of the output of $\tilde{\mathcal{C}}$. To enforce that, we will immediately subtract every value of $\tilde{\mathbf{x}}$ with its corresponding output (in a similar fashion with [18, 19, 21]). In this way, the input $\tilde{\mathbf{x}}$ corresponds to a correct computation transcript of \mathcal{C} if and only if $\tilde{\mathcal{C}}$ outputs a zero vector. What remains is to ensure that the prover uses the correct input \mathbf{V}_d . We can enforce the latter by appending \mathbf{V}_d in $\tilde{\mathbf{x}}$ and adding copy constraints. Finally we can naturally extend the same ideas for creating circuits of larger depth. For ease of exposition, we omit the details.

Proof of Lemma 1. We will prove Lemma 1 based on the above transformation. Observe that $\tilde{\mathcal{C}}$ contains all sub-circuits of \mathcal{C} , and for every element of $\tilde{\mathbf{x}}$, one additional gate. So $|\tilde{\mathcal{C}}| = 2|\mathcal{C}|$. $\tilde{\mathcal{C}}$ is data-parallel as it consists of multiple copies of d distinct 1-layered circuits that work on different inputs.

To argue about the space requirements of $\tilde{\mathbf{x}}$, we need to show how to instantiate access to $S(\tilde{\mathbf{x}})$.

Observe that $\tilde{\mathbf{x}}$ places the nodes of $\mathcal{T}_{\mathbf{V}}$ (i.e., sub-vectors) in post-order, which is the same order we follow when evaluating the circuit! Consequently, $S(\tilde{\mathbf{x}})$ will be a routine that follows the evaluation algorithm of \mathcal{C} and, on the j -th invocation, returns the newly computed node of $\tilde{\mathcal{T}}_{\mathbf{V}}$. Because $S(\tilde{\mathbf{x}})$ follows the evaluation algorithm, the time required for a complete scan over $\tilde{\mathbf{x}}$ is $|\mathcal{C}|$ field operations and $sp^S(\tilde{\mathbf{x}}) = |\mathbf{x}| + \sum_{i \in [d]} |\mathcal{C}'_i|$.

Finally, we need to prove that if \mathcal{C} is log-space uniform, then $\tilde{\mathcal{C}}$ is also log-space uniform. Recall that a circuit family is a log-space uniform if there exists a Turing Machine that on input 1^n and working space of $\mathcal{O}(\log n)$ outputs the description of the circuit [34]. Having that, we will show that if \mathcal{C} is log-space uniform, then we can create such a Turing machine M that uses the Turing machine for \mathcal{C} , M' , and outputs the description of $\tilde{\mathcal{C}}$. M uses the topological information of \mathcal{C} , namely the constant depth d , and $K_i, \forall i \in [d]$, invokes M' (possibly multiple times) and outputs the necessary information following a post-order traversal of $\mathcal{T}_{\mathcal{C}}$. Because d is constant and M' uses logarithmic space, it follows that M runs on logarithmic space too. So $\tilde{\mathcal{C}}$ is log-space uniform.

B.2 Proof of Theorem 2

Complexity. Let \mathcal{C} be an d -layered data parallel arithmetic circuit where d is a small constant (in case d is significantly larger than $\log \log |\mathcal{C}|$ we apply the transformation of Appendix B.1 to reduce the circuit's depth). Regarding prover complexity, Step 1 of **Construction 1** is dominated by $|\mathbf{w}|$ MSMs, step 2 needs $\mathcal{O}(|\mathcal{C}|(\log \log |\mathcal{C}| + d)) \approx \mathcal{O}(|\mathcal{C}| \log \log |\mathcal{C}|)$ field operations and step 3 requires $\mathcal{O}(|\mathbf{w}|)$ field operations and $\mathcal{O}(\sqrt{|\mathbf{w}|})$ MSMs/pairing operations. Overall, SPARROW requires $\mathcal{O}(|\mathcal{C}| \log \log |\mathcal{C}|)$ field operations and $\mathcal{O}(|\mathbf{w}|)$ MSMs.

Regarding space-complexity, steps 1 and 3 need space $\mathcal{O}(sp^S(\mathbf{w}) + \sqrt{|\mathbf{w}|})$ and step (2) needs space of $\mathcal{O}(|\mathbf{x}| + sp^S(\mathbf{w}) + \sum_{i \in [d]} |\mathcal{C}'_i| + \sqrt{|\mathcal{C}|})$. So, the overall space is $\mathcal{O}(|\mathbf{x}| + sp^S(\mathbf{w}) + \sum_{i \in [d]} |\mathcal{C}'_i| + \sqrt{|\mathcal{C}|})$. Observe that this space remains the same regardless of whether we apply the depth reduction.

As for the proof size, step 2 generates proofs of size $\mathcal{O}(d \log |\mathcal{C}|)$ and step 3 produces an evaluation proof of size $\mathcal{O}(\log |w|)$, so the overall proof size is $\mathcal{O}(d \log |\mathcal{C}|)$. If we apply depth reduction, the proof size becomes at most $\mathcal{O}(\log |\mathcal{C}|)$. In case \mathcal{C} is log-space uniform, verification time will be $\mathcal{O}(|\mathbf{x}| + \log |\mathcal{C}|)$.

Completeness. Derives from the completeness of our sumcheck protocol, space-efficient PC scheme, and the GKR protocol. Observe that replacing the standard sumcheck with our space-efficient sumcheck does not create any complications because, at the end of our sumcheck, the verifier ends up with two evaluations of the multi-linear polynomials f_i , as GKR requires.

Knowledge-Soundness. First, the soundness of the space-efficient variant of GKR comes directly from the soundness of our (information theoretic) sumcheck protocol A.3 and the soundness of the GKR protocol [34]. For knowledge-soundness, we can directly use the proof of [10] so we omit the details.

Zero-Knowledge. Derives from the zero-knowledge of our sumcheck protocol, GKR protocol, and PC scheme. To prove zero-knowledge, we use the same techniques as in [10], but with the difference that when proving zero-knowledge for the GKR protocol (Section 4.2 of [10]), the simulator makes a black-box invocation of the simulator of our sumcheck protocol.

C More details for our *zkFTP* Scheme

C.1 Training and Prediction Algorithms for Trees and Forests

Algorithm 3 gives a high-level description of the training forest training algorithm. More precisely, TRAIN (lines 18-24) takes as input the dataset $D \in [B]^{n \times d}$, the number of trees the forest consists of,

the number of features D contains, the maximum depth h of every tree in the forest, a random element seed needed to generate training randomness and returns the random forest \mathcal{F} . After initializing the necessary sets (line 19), we call BAGGING method that takes as input the dataset D and for every $i \in [K]$, sets $D_i \leftarrow \{\}$ and randomly picks n points in D using *seed*. Then for each $i \in [K]$, we invoke the TRAIN TREE method on input the dataset D_i and the attribute set \mathcal{A} , get the trained tree \mathcal{T}_i and update \mathcal{F} (lines 21-23). What remains is to explain TRAIN TREE— the training algorithm of a decision tree (lines 9-17).

First TRAIN TREE, tests if stopping criteria are met, e.g., whether the tree has depth h or the vast majority of elements in D have the same label (line 10). If these criteria hold, the algorithm stops a further split of the current node, computes its label (usually the most common label of its data), and assigns it as a leaf. Otherwise, we invoke FINDSPLIT (lines 1-8), which first computes the histograms for every feature and every class and then, using the histograms, finds and returns the best split. Having such a split, we split the dataset D in D_L and D_R (lines 12-14), remove the best feature from the set \mathcal{A} , create a new node (line 15), and recursively repeat the process for the left and right child of the node using D_L and D_R respectively (lines 16-17).

Algorithm 3 Forest Training Algorithm

```

1: procedure FINDSPLIT( $D, \mathcal{A}$ )
2:   Set  $H_{i,j} \leftarrow \mathbf{0}^B, \forall i \in [|\mathcal{A}|], j \in \{0, 1\}, \epsilon \leftarrow \{\}$ 
3:   for all  $p \in D$  and all  $i \in \mathcal{A}$  do
4:     if  $p.\text{label} = 1$  then  $H_{i,1}[p[i]]++$  else  $H_{i,0}[p[i]]++$ 
5:   for all  $i \in \mathcal{A}$  do
6:      $\epsilon \leftarrow \epsilon \cup (i, \text{BESTSPLITSCORE}(H_{i,0}, H_{i,1}))$ 
7:    $(j, \epsilon_j) \leftarrow \text{MIN}(\epsilon)$ 
8:   return  $(j, \epsilon_j)$ 

9: procedure TRAIN TREE( $\mathcal{T}, D, h, \mathcal{A}$ )
10:  if STOPPINGCRITERIA( $D, h$ ) then return
11:   $a, v \leftarrow \text{FINDSPLIT}(D, \mathcal{A})$ 
12:   $D_L = \{\}, D_R = \{\}$ 
13:  for all  $p \in D$  do
14:    if  $p[a] \leq v$  then  $D_L = D_L \cup p$  else  $D_R = D_R \cup p$ 
15:  ADDNODE( $\mathcal{T}, (a, v)$ )
16:  TRAIN( $\mathcal{T}, D_L, \mathcal{A} - \{a\}$ )
17:  TRAIN( $\mathcal{T}, D_R, \mathcal{A} - \{a\}$ )

18: procedure TRAIN( $D, K, d, h, \text{seed}$ )
19:   $\mathcal{F} \leftarrow \{\}, \{\mathcal{T}_i \leftarrow \{\}\}_{i \in [K]}, \mathcal{A} \leftarrow \{1, \dots, d\}$ 
20:  if  $K > 1$  then  $\{D_i\}_{i \in [K]} \leftarrow \text{BAGGING}(D, \text{seed})$ 
21:  for all  $i \in [K]$  do
22:    TRAIN TREE( $\mathcal{T}_i, D_i, \mathcal{A}$ )
23:     $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{T}_i$ 
24:  return  $\mathcal{F}$ 

```

Algorithm 4 provides a detailed description of the prediction process. Specifically, PREDICT (lines 9-14), takes as input the forest \mathcal{F} , a test point \mathbf{x} and for each tree in \mathcal{F} (lines 11-13), invokes the TREEPREDICT method (lines 1-8) and receives a prediction from every tree. Finally, return the majority of predictions \mathcal{Y} . Regarding TREEPREDICT, given a tree \mathcal{T} and a point \mathbf{x} , we start from the root and check if the root’s split value is larger than the point’s value at the split attribute. If

this is true, move to the left child. Otherwise, move to the right child. Repeat the same process until reaching a leaf, and return its id (i.e., relative position among other leaves) and the label (i.e., the prediction).

Algorithm 4 Forest Prediction Algorithm

```

1: procedure TREEPREDICT( $\mathcal{T}, \mathbf{x}$ )
2:    $node \leftarrow \mathcal{T}.root$ 
3:   while ISLEAF( $\mathcal{T}, n$ ) = 0 do
4:     if  $x[node.a] < node.v$  then
5:        $node \leftarrow node.left$ 
6:     else
7:        $node \leftarrow node.right$ 
8:   return  $node.leafID, node.label$ 
9: procedure PREDICT( $\mathcal{F} = \{T_i\}_{i \in [K]}, \mathbf{x}$ )
10:   $\mathcal{Y} \leftarrow \{\}$ 
11:  for all  $i \in [K]$  do
12:     $-, y \leftarrow \text{TREEPREDICT}(T_i, \mathbf{x})$ 
13:     $\mathcal{Y} \leftarrow \mathcal{Y} \cup y$ 
14:  return MAJORITY( $\mathcal{Y}$ )

```

C.2 Correctness Proof of our Certification Algorithm

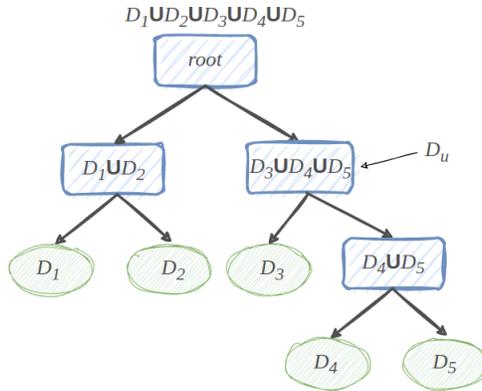


Figure 5: Example of a tree and the corresponding datasets for each node.

Claim 1 Let $D \in [B]^{n \times d}$ be a dataset, \mathcal{T} a decision tree, and h its maximum depth. If $\mathcal{T} \neq \text{TRAIN TREE}(D)$, then $\text{CERTIFY TREE}(\mathcal{T}, D)$ will always reject \mathcal{T} .

Proof. First, we denote with D_i (where $i \in [L]$) the set of all points such that $P[j] = i, \forall j \in [n]$ (computed following lines 13-15 of Algorithm 1). Figure 5 provides an example of a tree and its corresponding sets D_i . In addition, for every non-leaf node v , we represent with D_v all the points in D that pass through v when performing an inference step.

Next, from the properties of decision trees, it is not hard to see that (i) $D = \bigcup_{i \in [L]} D_i$ (ii) $D_i \cap D_j = \emptyset, \forall i, j \in [L]$ and $i \neq j$, (iii) $D_v = \bigcup_{i \in \mathcal{L}_v} D_i$, where \mathcal{L}_v is the set of leaves that have

v as ancestor and (iv) from the homomorphic properties of the histograms and observation (iii), $H = \text{Hist}(D_v) = \text{Hist}(\bigcup_{i \in \mathcal{L}_v} D_i) = \sum_{i \in \mathcal{L}_v} \text{Hist}(D_i)$.

Having established that, assume that \mathcal{T} is not well-formed. We will prove that CERTIFYTREE will always reject it (return 0). As a starting point, we will show that if the root node is incorrect, then Algorithm 1 will always detect the inconsistency. Let H_{root} , be the histograms computed by Algorithm 1, from lines 18-23. We can re-write H_{root} as $\sum_{i \in [L]} H_{i,\dots}^{leaf}$ and from observation (iv), we know that $\sum_{i \in [L]} H_{i,\dots}^{leaf} = \text{Hist}(\bigcup_{i \in [L]} D_i)$. Moreover, from observation (iii), we have $\text{Hist}(\bigcup_{i \in [L]} D_i) = \text{Hist}(D)$. Consequently, the first invocation of FINDSPLIT method of TRAI NTREE algorithm (Algorithm 3) and VALIDATESPLIT of Algorithm 1 must compute the same split value and feature. Thus, if \mathcal{T} has a different split value or feature, then it is incorrect and will be rejected according to VALIDATESPLIT method.

Now, assuming that the root is consistent, we need will show that the histograms of its children computed in Algorithm 3 are the same as the ones computed in Algorithm 1. Let D_L, D_R be the data that correspond to the left and right children of the root as computed at line 14 of Algorithm 3. Furthermore, assume that $D'_L = \bigcup_{i \in [s]} D_i$ (and $D'_R = \bigcup_{i \in \{s+1, \dots, L\}} D_i$) be the set points that pass through the left (and right resp.) child of the root node of the tree (for instance in Figure 5, $D'_L = D_1 \cup D_2$ and $D'_R = D_3 \cup D_4 \cup D_5$). We first need to prove that $D'_L = D_L$ (and $D'_R = D_R$ resp.). Because the root node is correct, for all points $p \in D'_L$, we will have that $p[\text{root}.a] \leq \text{root}.v$. In addition, due to the fact that $D'_L \cup D'_R = D$ (from (i)) and $D'_L \cap D'_R = \emptyset$ (from (ii)), we conclude that $D'_L = D_L$.

Finally, from (iv) we know that $\sum_{i \in [s]} H_{i,\dots}^{leaf} = \text{Hist}(D'_L) = \text{Hist}(D_L)$ (and resp. for D_R). Having established the correctness of the histograms of the children, we can follow the same argument as we did for the root recursively until reaching the leaves.

C.3 Correctness Proofs of Histogram and Tree Computations Circuits

Permutation check Consider the sets A, B , and let $a_i \in \mathbb{F}^n$ denote an element of A (and b_i resp. for B). To prove that A is a permutation of B , the we need to show that for a randomly selected point $s \in \mathbb{F}$, $\prod_{i \in [A]} (\langle (1, s, \dots, s^n), a_i \rangle + r^{n+1}) = \prod_{i \in [A]} (\langle (1, s, \dots, s^n), b_i \rangle + r^{n+1})$. Due to Schwartz–Zippel lemma, the probability that the above equality holds but $A \neq B$ is $\frac{|A|n}{|\mathbb{F}|}$. To compile this check into an arithmetic circuit, we use two multiplication trees that take as input $1, s, \dots, s^n, s^{n+1}, A, B$ and outputs the product difference.

Claim 2 Given a vector of addresses $\mathbf{h} \in [B]^N$, read/write transcripts $H_r, H_w \in ([B], \mathbb{N})^N$ such that $\forall i \in [N]$, (i) $H_r[i][1] + 1 = H_w[i][1]$ and (ii) $H_r[i][0] = H_w[i][0] = \mathbf{h}_i$, then $H \in [\mathbb{N}]^B$ is a histogram of \mathbf{h} (i.e., $\forall i \in [B], H[i] = \sum_{j \in [N]} I(\mathbf{h}_j = i)$) if and only if $H_w \cup \{(i, 0)\}_{i \in [B]} = H_r \cup H$.

Proof. We denote with H_r^i (and H_w^i resp.) the set that contains all pairs of the form $\{(i, v_j)\}_{j \in [u]}$ (and $\{(i, v'_j)\}_{j \in [u]}$ resp.), where $u = \sum_{j \in [N]} I(\mathbf{h}_j = i)$. Note that due to conditions (i)-(ii), it will always hold that $|H_r^i| = |H_w^i| = u$ and $v_j + 1 = v'_j, \forall j \in [u]$.

Having established that, assume that exists an address $i \in [B]$, such that $H[i] = u' \neq u$ but still $H_b \cup H_w = H_r \cup H$. We will reach a contradiction. First, because i is distinct and from condition (ii), it must also hold that $H_w^i \cup (i, 0) = H_r^i \cup (i, u')$, so it is enough to focus on this subset. We now consider two cases. Assume that $u' > u$. Then (i, u') must belong to H_w^i and thus due to condition (i), $(i, u' - 1)$ must belong to H_r^i . We continue recursively the same argument for $(i, u' - 1)$, until reaching $(i, u' - u)$. At this point $(i, u' - u)$ (where $u' - u > 0$), must belong to H_w^i . However $|H_w^i| = u$ meaning that either $(i, u' - u)$ or a pair between (i, u') and $(i, u' - u)$ does not belong to H_w^i , so $H_w^i \cup (i, 0) \neq H_r^i \cup (i, u')$, or $|H_w^i| > u$ which contradicts condition (ii).

In the case where $u' < u$, we follow a similar argument. To begin with, observe that if there exists a pair (i, u'') in H_w^i , such that $u'' > u'$, then due to condition (i), (i, u'') will not exist in $H_r^i \cup (i, u')$, so $H_w^i \cup (i, 0) \neq H_r^i \cup (i, u'')$. Moreover, if H_r^i has some negative values, then due to condition (i), there will be a pair $(i, u'') \in H_r^i$, s.t $u'' < 0$ and $(i, u'') \neq H_w^i$ (i.e., u'' is the minimum value of H_r^i) so $H_w^i \cup (i, 0) \neq H_r^i \cup (i, u')$. Consequently, we limit the proof in the case that all positive values $u'' \leq u'$. Due to the pigeonhole principle there must be in H_r^i at least two pairs of the form $(i, v_j), (i, v_{j'})$ where $v_j = v_{j'}$. If there is a smaller number of (i, v_j) pairs in H_w^i , then $H_w^i \cup (i, 0) \neq H_r^i \cup (i, u')$. Otherwise, there must be the same number of pairs in H_r^i of the form $(i, v_j - 1)$. We follow the same argument recursively until having multiple pairs of the form $(i, 0)$. In this case, $H_w^i \cup (i, 0) \neq H_r^i \cup (i, u')$, as the left side set must contain only one pair of the form $(i, 0)$.

Claim 3 Given \mathcal{L} , a set containing the coordinates of the leaves along with their data and \mathcal{S} the tree computation transcript and \mathcal{N} the set such that $\forall i \in \llbracket |\mathcal{S}|/2 \rrbracket$, $\mathcal{N}_i = (h_{2i} - 1, p_{2i}/2, \mathcal{G}(D_{2i}, D_{2i+1}))$ (where $h_{2i} = h_{2i+1}$ and $p_{2i} = p_{2i+1} + 1$), then \mathcal{S} is consistent if and only if $\mathcal{N} \cup \mathcal{L} = \mathcal{S} \cup (0, 0, D_r)$.

Proof. To begin with, we say that \mathcal{S} is inconsistent if (i) v_l, v_r are siblings but their parent (if it is not a root) is not encoded in \mathcal{S} , (ii) there exists a leaf, not included in \mathcal{S} . If these conditions do not hold, then \mathcal{S} is a valid transcript of a computation over a tree. To see why this is the case let \mathcal{S}_1 be a set containing all pairs of leaves and their siblings. Because (ii) does not hold, then $\mathcal{S}_1 \subset \mathcal{S}$. Let \mathcal{S}_2 be the pairs of parent nodes (along with their siblings) computed by \mathcal{S}_1 . Because (i) does not hold, we know that $\mathcal{S}_2 \subset \mathcal{S}$. We apply the same argument recursively until reaching the root. In this way, \mathcal{S} contains all nodes of the tree and their correctly computed data, so it is a consistent computation transcript.

What remains is to show that if (i) or (ii) hold then, $\mathcal{N} \cup \mathcal{L} \neq \mathcal{S} \cup (0, 0, D_r)$. If (ii) holds, then there is a tuple $l \in \mathcal{L}$ but not in \mathcal{S} , so $\mathcal{N} \cup \mathcal{L} \neq \mathcal{S} \cup (0, 0, D_r)$. If (i) holds, then there will be a pair $(h, p, D_v) \in \mathcal{N}$ that does not belong to \mathcal{S} so $\mathcal{N} \cup \mathcal{L} \neq \mathcal{S} \cup (0, 0, D_r)$.

Claim 4 Given a vector of addresses $\mathbf{h} \in [B]^N$, frequencies $\mathbf{F}_B \in \mathcal{N}^N$, read/write transcripts $H_r, H_w \in ([B], \mathbb{N})^N$ such that $\forall i \in [N]$, (i) $H_r[i][1] + \mathbf{F}_B[i] = H_w[i][1]$ and (ii) $H_r[i][0] = H_w[i][0] = \mathbf{h}_i$, then $H \in [\mathbb{N}]^B$ is a histogram of \mathbf{h} (i.e., $\forall i \in [B], H[i] = \sum_{j \in [N]} \mathbf{F}_B[j] I(\mathbf{h}_j = i)$) if and only if $H_w \cup \{(i, 0)\}_{i \in [B]} = H_r \cup H$.

Proof. Our proof follows a similar argument with Claim 2. Focusing on a single address $i \in [B]$, let $(k_1, \dots, k_{|H_w^i|})$, be the set of indexes such that $I(\mathbf{h}_{k_j} = i) = 1$ and let $H[i] = u' \neq u = \sum_{j \in |H_w^i|} \mathbf{F}_B[k_j]$, but still $H_w^i \cup (i, 0) = H_r^i \cup (i, u')$. We will reach a contradiction. First observe that (i, u') must belong to H_w^i and thus due to condition (i), $(i, u' - \mathbf{F}_B[k_1])$ must belong to H_r^i . Similarly, $(i, u' - \mathbf{F}_B[k_1])$ must belong to H_w^i . We repeat the same argument, $|H_w^i|$ times, concluding that $(i, u' - \sum_{i \in |H_w^i|} \mathbf{F}_B[k_i])$ must belong to H_r^i . Because $u' - \sum_{i \in |H_w^i|} \mathbf{F}_B[k_i] = u' - u \neq 0$, then clearly $(i, u' - \sum_{i \in |H_w^i|} \mathbf{F}_B[k_i]) \neq (i, 0)$, so $H_w^i \cup (i, 0) \neq H_r^i \cup (i, u')$.

C.4 Circuit that Validates Correctness of Inference [39]

Let T , be a tree path that consists of the leaf index, its relative position in the tree and a list of triples, one for each node of the tree from the leaf to the root, containing the feature id $f_i \in [d]$, the split value $v_i \in [B]$ and the direction $d_i \in \{0, 1\}$ (e.g., 0 if we move to the left child or 1 otherwise). Note that we pad the list of triples to have the maximum depth of the tree. Overall we represent the path as $T = ((f_1, v_1, d_1), \dots, (f_{max}, v_{max}, d_{max}), (L_i, pos_i))$. Having established that, we construct a circuit that takes as input T , the test point $\mathbf{x} = ((1, x_1), (2, x_2), \dots, (d, x_d))$, represented by the attribute id and

its corresponding value, and a permuted test point for $\tilde{\mathbf{x}} = ((a[1], x_{a[1]}), (a[2], x_{a[2]}), \dots, (a[d], x_{a[d]}))$, where $a : [d] \rightarrow [d]$ is a permutation s.t $\forall i \in [max], a[i] = f_i$. Next the circuit checks (i) if the attributes in the permuted point are consistent with that of T , (e.g., $\tilde{\mathbf{x}}[i][0] - T[i][0] = 0$) and (ii) $\tilde{\mathbf{x}}$ is a permutation of \mathbf{x} . The last check is achieved by the permutation circuit (see Checking set equality) of C.3. Finally, the circuit needs to check that the split was performed correctly. To achieve that, it computes the difference of the values first max values of $\tilde{\mathbf{x}}$ and T (e.g., $\tilde{\mathbf{x}}[i][1] - T[i][1] = 0, \forall i \in [max]$). If the difference is negative and the direction is zero, or vice-versa, reject. Otherwise output 1.

C.5 Space Complexity of Streaming Oracles

In this section we analyze the space required to instantiate streaming access to the input of every circuit of the proving training algorithm.

(0) Checking Bagging Correctness. To instantiate $S(\mathbf{g})$ we need to store a single element \mathbf{g}_{i-1} used to compute \mathbf{g}_i , and also $A, B, seed$, leading to $sp^S(\mathbf{g}) = 4$. For $S(\mathbf{F}_B)$ we have $sp^S(\mathbf{F}_B) = |T_r| + sp^S(\mathbf{g})$. Because the input streams of the circuits use $S(\mathbf{g}), S(\mathbf{F}_B)$, their required space will be the same but with some additive constant overhead.

(1) Checking Correctness of Assignments. Starting with the simple case where $K = 1$, it is not hard to see that the space required to instantiate both oracles is $sp^S(\cdot) = |D| + |T| + \mathcal{O}(d)$ as they only require access to $S(D)$, the forest, and some buffer space of size $\mathcal{O}(d)$ to compute the auxiliary information. For the general case where $K > 1$, the space needed to instantiate access to these oracles is identical to the case where $K = 1$ —independent of K !

(2) Checking Leaf Histograms. For $K = 1$, note that $sp^S(\mathbf{h}) = |D| + sp^S(T') = |D| + |T| + \mathcal{O}(d)$ while $sp^S(H_{w,r}) = sp^S(\mathbf{h}) + dLB = |D| + |T| + dLB + \mathcal{O}(d)$. Finally, $sp^S(H^L) = |D| + dLB + |T|$. Likewise with step (1), the space needed for $K > 1$ is independent of K .

(3) Checking Non-Leaf Histograms. In the simple case where $K = 1$, we do not have to instantiate any streaming oracle. When $K > 1$, we need to construct $S(H)$, which on j -th invocation, calls $S(H^L)$ to get $H^{L,j}$ and uses it to compute and output H^j . So the required space is $sp^S(H) = dLB + sp^S(H^L) = |D| + 2dLB + |T|$.

(4) Checking Non-Leaf Histograms. We can directly prove the correctness of splits by using $sp^S(H)$ and generating “on the fly” all the data we need (e.g., gini indexes, bit-decomposition values from comparisons, etc.). In practice, the overall space required is no more than $sp^S(H) + dLB$.

C.6 Proof of Theorem 3

Training Complexity. Regarding proving time, observe that the largest circuit size is $\mathcal{O}(|D|K)$ (Circuit \mathcal{C}_3 of **Construction 1**). Because we invoke SPARROW prover only a constant number of times, the total proving complexity will be $\mathcal{O}(|D|K \log \log(|D|K))$. As for proof size, because we have a constant number of proofs, due to Theorem 2, we will have a proof size of $\mathcal{O}(\log K |D|)$. Similarly, with verification time, we also observe that all circuits are data-parallel with sub-circuits of small constant size (e.g., circuits \mathcal{C}_3, Δ) or multiplication trees (e.g., $\mathcal{C}_1, \mathcal{C}_5$) which are inherently log-space uniform. So, from Theorem 2, we have verification time of $\mathcal{O}(\log K |D|)$.

To argue about space complexity, note that every invocation of SPARROW has different space requirements that depend on the structure of the circuit and the space necessary to instantiate the streaming oracles. We will give an upper bound by measuring the largest space required for streaming oracles, for evaluating a circuit, and the maximum circuit size. From Section C.5, the we get a maximum space of $sp^S(\cdot) = |D| + |\mathbf{F}| + 2dLB$, for evaluation we get dLB and for the circuit

size we have $|C_3| = c|D|K$, where $c \in \mathbb{N}$ is a large multiplicative constant. Thus, the concrete space complexity of the prover will be $|D| + |\mathbf{F}| + 3dLB + \sqrt{c|D|K}$ field elements.

Training Completeness. Training completeness comes directly from the completeness of SPARROW, standard GKR-based SNARK [10] and Claims 1,2,4,3.

Proof Sketch of Forest and Data Extractability. Suppose that an adversary \mathcal{A} against the forest and data extractability game convinces a verifier with non-negligible probability. We will create an extractor \mathcal{E} , which having access to the random tape of \mathcal{A} and the public parameters, extracts a dataset D and a correctly trained forest (over the dataset and a seed).

To do so, we will construct ten adversaries $\mathcal{A}_1, \dots, \mathcal{A}_{10}$ (one for every proof) against the knowledge soundness game of GKR-based SNARKs (both standard and space-efficient). All adversaries are given the public parameters, have access to the random tape of \mathcal{A} , and output the following:

- \mathcal{A}_1 : Returns the proof π_1 of step 1.b.i, including the PC commitment $C_{\mathbf{g}}$.
- \mathcal{A}_2 : Returns the proof π_2 of step 1.b.ii including $C_{\mathbf{g}}^{(2)}, C_{\mathbf{F}_B}$.
- \mathcal{A}_3 : Returns the proof π_3 of step 2.b.i including $C_{T'}$.
- \mathcal{A}_4 : Returns the proof π_4 of step 2.b.ii including $C_{\mathcal{F}}$.
- \mathcal{A}_5 : Returns the proof π_5 of step 2.b.iii which includes $C_D, C_{T'}^{(2)}$.
- \mathcal{A}_6 : Returns the proof π_6 of 3.b.i including $C_{\mathbf{h}}, C_{\mathbf{H}_r}, C_{\mathbf{H}_w}, C_{\mathbf{F}_B}^{(2)}$.
- \mathcal{A}_7 : Returns the proof π_7 of step 3.b.ii including $C_{H^L}, C_{\mathbf{H}_r}^{(2)}, C_{\mathbf{H}_w}^{(2)}$.
- \mathcal{A}_8 : Returns the proof π_8 of step 4.b including $C_{\mathcal{F}}^{(2)}, C_{H^L}^{(2)}, C_H$.
- \mathcal{A}_9 : Returns the proof π_9 of step 5.a including $C_{\mathcal{F}}^{(3)}, C'_H$.
- \mathcal{A}_{10} : Returns the proof π_{10} of step 6 including $C_{\mathcal{F}}^{(4)}$.

Then, we invoke the ten extractors \mathcal{E}_i , one for every adversary, that take the same input as \mathcal{A}_i and output the following:

- \mathcal{E}_1 : Outputs $\mathbf{w} = (\mathbf{w}_1 = \mathbf{g}, \mathbf{w}_2)$ such that $\mathcal{C}_{B,1}(\mathbf{x}, \mathbf{w}) = 1$ and \mathbf{g} is the pre-image of $C_{\mathbf{g}}$ with overwhelming probability due to Theorem 2.
- \mathcal{E}_2 : Outputs $\mathbf{w} = (\mathbf{w}_1 = \mathbf{g}^{(2)}, \mathbf{w}_2 = \mathbf{F}_B, \mathbf{w}_3)$ such that $\mathcal{C}_{B,2}(\mathbf{x}, \mathbf{w}) = 1$, \mathbf{g} is the pre-image of $C_{\mathbf{g}}^{(2)}$ and \mathbf{F}_B is the pre-image of $C_{\mathbf{F}_B}$ with overwhelming probability due to Theorem 2.
- \mathcal{E}_3 : Outputs $\mathbf{w} = T'$ such that $\mathcal{C}_1(\mathbf{x}, \mathbf{w}) = 1$ and T' is the pre-image of $C_{T'}$ with overwhelming probability due to Theorem 2.
- \mathcal{E}_4 : Outputs $\mathbf{w} = \mathbf{F}$ such that $\mathcal{C}_2(\mathbf{x}, \mathbf{w}) = 1$ and \mathbf{F} is the pre-image of $C_{\mathcal{F}}$ with overwhelming probability due to Theorem 6 of [10].
- \mathcal{E}_5 : Outputs $\mathbf{w} = (\mathbf{w}_1 = D, \mathbf{w}_2 = T'^{(2)}, \mathbf{w}_3)$ such that $\mathcal{C}_3(\mathbf{x}, \mathbf{w}) = 1$, $D, T'^{(2)}$ are the pre-images of $C_D, C_{T'}^{(2)}$ respectively, with overwhelming probability due to Theorem 2.
- \mathcal{E}_6 : Outputs $\mathbf{w} = (\mathbf{w}_1 = \mathbf{h}, \mathbf{w}_2 = \mathbf{H}_r, \mathbf{w}_3 = \mathbf{H}_w, \mathbf{w}_4 = \mathbf{F}_B^{(2)})$ such that $\mathcal{C}_4(\mathbf{x}, \mathbf{w}) = 1$, $\mathbf{h}, \mathbf{H}_r, \mathbf{H}_w, \mathbf{F}_B^{(2)}$ are the pre-images of $C_{\mathbf{h}}, C_{\mathbf{H}_r}, C_{\mathbf{H}_w}$ and $C_{\mathbf{F}_B}^{(2)}$ respectively, with overwhelming probability due to Theorem 2.
- \mathcal{E}_7 : Outputs $\mathbf{w} = (\mathbf{w}_1 = \mathbf{h}^{(2)}, \mathbf{w}_2 = \mathbf{H}_r^{(2)}, \mathbf{w}_3 = \mathbf{H}_w^{(2)}, \mathbf{w}_4 = H^L)$ such that $\mathcal{C}_5(\mathbf{x}, \mathbf{w}) = 1$, $\mathbf{H}_r^{(2)}, \mathbf{H}_w^{(2)}, H^L$ are the pre-images of $C_{\mathbf{H}_r}^{(2)}, C_{\mathbf{H}_w}^{(2)}$ and C_{H^L} , with overwhelming probability due to Theorem 2.
- \mathcal{E}_8 : Outputs $\mathbf{w} = (\mathbf{w}_1 = H^{L,(2)}, \mathbf{w}_2 = H, \mathbf{w}_3 = \mathbf{F}^{(2)}, \mathbf{w}_4)$ such that $\mathcal{C}_6(\mathbf{x}, \mathbf{w}) = 1$, $H^{L,(2)}, H, \mathbf{F}^{(2)}$ are the pre-images of $C_{H^L}^{(2)}, C_H$ and $C_{\mathcal{F}}^{(2)}$ respectively, with overwhelming probability due to Theorem 2.
- \mathcal{E}_9 : Outputs $\mathbf{w} = (\mathbf{w}_1 = H, \mathbf{w}_2 = \mathbf{F}^{(3)}, \mathbf{w}_3)$ such that $\mathcal{C}_6(\mathbf{x}, \mathbf{w}) = 1$, $H^{(2)}, \mathbf{F}^{(3)}$ is the pre-image of $C_H^{(2)}$ and $C_{\mathcal{F}}^{(3)}$ respectively, with overwhelming probability due to Theorem 2.

- \mathcal{E}_{10} : Outputs $\mathbf{w} = (\mathbf{w}_1 = F^{(4)}, \mathbf{w}_2)$ such that $\mathcal{C}_8(\mathbf{x}, \mathbf{w}) = 1$, and $F^{(4)}$ is the pre-image of $C_F^{(4)}$ with overwhelming probability due to Theorem 6 of [10].

At this point, observe that $C_{\mathbf{g}} = C_{\mathbf{g}}^{(2)}$, $C_{\mathbf{F}_B} = C_{\mathbf{F}_B}^{(2)}$, $C_{T'} = C_{T'}^{(2)}$, $C_{\mathbf{h}} = C_{\mathbf{h}}^{(2)}$, $C_{\mathbf{H}_r} = C_{\mathbf{H}_r}^{(2)}$, $C_{\mathbf{H}_w} = C_{\mathbf{H}_w}^{(2)}$, $C_{H^L} = C_{H^L}^{(2)}$, $C_H = C_H^{(2)}$, $C_{\mathcal{F}} = C_{\mathcal{F}}^{(2)} = C_{\mathcal{F}}^{(3)} = C_{\mathcal{F}}^{(4)}$ — otherwise the verification algorithm would reject. Consequently, due to the binding properties of PC schemes, we have $\mathbf{g} = \mathbf{g}^{(2)}$, $\mathbf{F}_B = \mathbf{F}_B^{(2)}$, $T' = T'^{(2)}$, $\mathbf{h} = \mathbf{h}^{(2)}$, $\mathbf{H}_r = \mathbf{H}_r^{(2)}$, $\mathbf{H}_w = \mathbf{H}_w^{(2)}$, $H^L = H^{L,(2)}$, $H = H^{(2)}$, $\mathbf{F} = \mathbf{F}^{(2)} = \mathbf{F}^{(3)} = \mathbf{F}^{(4)}$. Finally \mathcal{E} , returns D, \mathbf{F} .

What remains, is to show that $\mathcal{F} = \text{Train}(D, \text{seed})$ (\mathbf{F} the matrix encoding of \mathcal{F}) with overwhelming probability. Due to the soundness of GKR protocol, we know that \mathbf{g} is a vector containing values following a uniform-like distribution generated by a Linear congruential generator. Furthermore, from [89] and the soundness of GKR, we know that $\mathbf{F}_B[i] = T_r[\mathbf{g}_i]$ with overwhelming probability. Next, \mathcal{C}_1 and \mathcal{C}_2 guarantee that the rows of T' belong to \mathbf{F}^T with overwhelming probability and \mathcal{C}_3 guarantees that that T' is well formed—namely for every p_i , every j -th tree $j \in [K]$, $T_i'^j$ is the correct path of p_i in tree j . Next, \mathcal{C}_4 guarantees with overwhelming probability that $\mathbf{H}_r^j[i] = \mathbf{H}_w^j[i][0] = \mathbf{h}^j[i][0]$ and $\mathbf{H}_r^j[i][1] + \mathbf{F}_B^j[i] = \mathbf{H}_w^j[i][1]$. Moreover, \mathcal{C}_5 guarantees that $\mathbf{H}_w \cup H_{init} = \mathbf{H}_r \cup H^L$ with overwhelming probability, thus from Claim 4 we know that H^L contains correctly computed histograms. For non-leaf histograms, we first parse the witness of \mathcal{C}_6 to get the computation transcript \mathcal{S} . Because \mathcal{C}_6 validates that $H^{L*} \cup H^* = \mathcal{S} \cup (0, 0, H_r)$ with overwhelming probability, from Claim 3 we know that H is the correctly computed set of non-leaf histograms and from **Step 5**, these histograms will result in the correct splits. Finally, from **Step 6** we know that \mathbf{F} represents a valid forest.

Note that these checks, correspond to the computations the certification algorithm (see Algorithm 1) needs to apply. Because the algorithm accepts the forest, then from Claim 1, this forest must be correctly trained.

Last but not least, because \mathcal{E} invokes the extractors $\mathcal{E}_1, \dots, \mathcal{E}_{10}$, the probability that \mathcal{E} fails is bounded by the probabilities the GKR-based extractors fail, which is negligible. Furthermore, because all \mathcal{E}_i run in polynomial time, \mathcal{E} will run in polynomial time too.

Proof Sketch for Training Zero-Knowledge. To prove zero-knowledge, we will construct a simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$, that takes the public parameters \mathbf{pk}, \mathbf{vk} and trapdoor information \mathbf{trap} which will be used by the simulator of the PC scheme to generate accepting evaluation proofs. At a high level, our Simulator receives training parameters from the adversary and interacts with it following the same procedure as in **Construction 1**, but instead of invoking the commit and generating SPARROW or standard GKR-based proofs, calls their corresponding simulators. In more detail, \mathcal{S} works as follows:

1. Receive n, d, h, K, seed from \mathcal{A} .
2. $C_D \leftarrow \mathcal{S}_1(\mathbf{pk}, \mathbf{trap}, n, d)$: Invokes the simulator of the PC scheme (\mathcal{S}_{PC}) providing him with the public parameters, and the trapdoor information and returns the output of \mathcal{S}_{PC} .
3. $C_{\mathcal{F}} \leftarrow \mathcal{S}_2(\mathbf{pk}, \mathbf{trap}, K, h, \text{seed})$: \mathcal{S}_2 works in a similar fashion as with \mathcal{S}_1 .
4. $\pi_T \leftarrow \mathcal{S}_3(\mathbf{pk}, \mathbf{trap}, C_D, C_{\mathcal{F}}, \text{seed})$: In the **Commit** phase of **Construction 3**, \mathcal{S}_3 , invokes \mathcal{S}_{PC} , and receives the necessary commitments. Next, it invokes ten simulators $\mathcal{S}_{SNARK}^{(i)}$, of the GKR-based zkSNARK scheme, one for every step of **Construction 3**. Finally, \mathcal{S}_3 outputs $\{\{C_{\mathbf{F}_B}^*, C_{T'}^*, C_{H^L}^*, C_H^*\}, \{\pi_i^*\}_{i \in [10]}\}$ providing them the necessary trapdoor information so that they can generate a valid evaluation proof.

What remains is to prove that π_T^* , the proof \mathcal{S} outputs, is indistinguishable from the proof π_T generated by an honest PPT prover. Because (i) the commitments received by the adversary are indistinguishable due to the hiding properties of the PC scheme and (ii) the proofs $\{\pi_i\}_{i \in [10]}$ are indistinguishable with the proofs $\{\pi_i^*\}_{i \in [10]}$ due to the zero-knowledge property of the GKR-based SNARK [10] and SPARROW³, it follows that π_T^* will be indistinguishable to π_T for any PPT adversary \mathcal{A} .

Formally, we can use the standard hybrid argument, consisting of three hybrids. The first hybrid (H_0) outputs the proof in which the prover behaves honestly following **Construction 2** and **Construction 3**. The second hybrid (H_1) replaces all the invocations of the zkSNARK provers with their Simulators, while the third hybrid follows the same steps with the simulator S . Observe that due to the zero-knowledge properties of the underlying zkSNARK protocols, a PPT adversary cannot distinguish H_0 and H_1 . Moreover, due to the hiding properties of the PC scheme, an adversary cannot distinguish H_1 and H_2 . So, any PPT adversary fails to distinguish between H_0 and H_2 .

Proof Sketch of Forest Extractability. Consider a PPT adversary \mathcal{A} that wins the forest extractability game with non-negligible probability. We will create a PPT extractor \mathcal{E} that has access to the random tape of \mathcal{A} , the public parameters and extracts a forest \mathcal{F} such that on input the test point \mathbf{x} , predicts y with overwhelming probability.

First we construct the adversaries $\mathcal{A}_{link}, \mathcal{A}_{lkp}, \mathcal{A}_{pred}$, against the knowledge soundness of the linking, matrix lookup (e.g., the relation \mathcal{R}_{lkp}) and CP-SNARK (e.g., the relation \mathcal{R}_{pred}) protocols respectively. All adversaries have access to the random tape of \mathcal{A} and output the following:

- \mathcal{A}_{link} : Returns the proof π_{link} of step 1.b, in *ProvePred*, **Construction 2** consisting of two sumcheck proofs (one to prove $p_{\mathbf{F}}(s) = \sum_{x \in \{0,1\}^{\log KLp}} f_{\mathbf{y}}(x)g_{\mathbf{F}}(x)$ and one to prove the well-formedness of $f_{\mathbf{y}}(x)$), the PC commitment $C_{\mathcal{F}}^T$ the KZG commitment $C_{\mathcal{F}}^P$ and their corresponding evaluation proofs.
- \mathcal{A}_{lkp} : Returns the proof π_{lkp} of step 3, in *ProvePred*, **Construction 2** which also includes the commitments $C_{\mathbf{P}}, C_{\mathcal{F}}^{P(2)}$.
- \mathcal{A}_{pred} : Returns the proof π_{pred} of step 4, in *ProvePred*, **Construction 2** which also includes the commitment $C_{\mathbf{P}}^{(2)}$.

Then, we invoke the extractors $\mathcal{E}_{link}, \mathcal{E}_{lkp}, \mathcal{E}_{pred}$ which take the same input as their corresponding adversaries and output the following:

- \mathcal{E}_{link} : Internally invokes the extractors of the PC scheme and returns the univariate polynomial $p(x)$ and the multivariate one $g_{\mathbf{F}}(x)$ such that $p_{\mathbf{F}}(s) = \sum_{x \in \{0,1\}^{\log KLp}} f_{\mathbf{y}}(x)g_{\mathbf{F}}(x)$ with overwhelming probability.
- \mathcal{A}_{lkp} : Outputs the univariate polynomial $p^{(2)}(x)$, the path \mathbf{P} and auxiliary witness information, satisfying the relation \mathcal{R}_{lkp} .
- \mathcal{A}_{pred} : Outputs the path $\mathbf{P}^{(2)}$ and auxiliary information, satisfying the relation \mathcal{R}_{pred} .

For any accepting proof \mathcal{A} generates, it must hold that $C_{\mathcal{F}}^P = C_{\mathcal{F}}^{P(2)}$ and $C_{\mathbf{P}} = C_{\mathbf{P}}^{(2)}$. Thus, due to the binding properties of the underlying commitments, it should hold that $p(x) = p^{(2)}(x)$, $g_{\mathbf{F}}(x) = g_{\mathbf{F}}^{(2)}(x)$ and $\mathbf{P}^{(2)} = \mathbf{P}$. Furthermore, due to the knowledge soundness of matrix-lookup argument [40] it holds that \mathbf{P} which is the pre-image of $C_{\mathbf{P}}$ and that every row of \mathbf{P} belongs to \mathbf{F} with overwhelming probability. In addition, due to the knowledge soundness properties of the underlying CP-SNARK scheme [9], we know that each row of \mathbf{P} belongs to a different tree in the forest, and that it is a valid prediction path with overwhelming probability. What remains is to show that $p(s) \neq \sum_{x \in \{0,1\}^{\log KLp}} f_{\mathbf{y}}(x)g_{\mathbf{F}}(x)$ with negligible probability. This derives naturally from the

³Note that some proofs share common commitments, we appropriately increase the degree of the masking polynomial applied to the zero-knowledge PC scheme so that the evaluation proofs will be indistinguishable.

soundness of the sumcheck protocol and the polynomial commitment scheme. Finally, the coefficients of $p(x)$ and $g_{\mathbf{F}}(x)$ are the same, corresponding to an encoding \mathbf{F} of a tree \mathcal{F} with probability $1 - \frac{K L p}{|\mathbb{F}|}$ due to the Schwartz-Zippel Lemma.

Finally, the probability \mathcal{E} fails, is bounded by the probability extractors $\mathcal{E}_{link}, \mathcal{E}_{lkp}, \mathcal{E}_{pred}$ fail which is negligible. In addition the running time of our extractor is the sum of the running times of $\mathcal{E}_{link}, \mathcal{E}_{lkp}, \mathcal{E}_{pred}$. Because the latter is polynomial time, \mathcal{E} will run in polynomial time. We thus conclude that the extractor returns a forest \mathcal{F} that predicts y on \mathbf{x} with overwhelming probability.

Proof Sketch for Prediction Zero-Knowledge. Zero-Knowledge comes directly from the zero-knowledge of our sumcheck based zkSNARK for the linking protocol, the zero-knowledge of the matrix lookup argument and the zero-knowledge of the underlying CP-SNARK scheme. More precisely, we create a simulator \mathcal{S} that takes as input a maximum height h , number of trees K and uses the simulator of the PC scheme \mathcal{S}_{PC} , the linking protocol \mathcal{S}_{link} , lookup argument \mathcal{S}_{lkp} , the CP-SNARK \mathcal{S}_{pred} and works as follows:

1. Invokes the simulator \mathcal{S}_{PC} giving providing him with the public parameters and returns its output which is the commitment $C_{\mathcal{F}}^* = \{C_{\mathcal{F}}^T, C_{\mathcal{F}}^P\}$.
2. Receives \mathbf{x} from the adversary.
3. Invoke the simulator for the sumcheck based SNARK for the linking protocol \mathcal{S}_{link} giving him the public parameters, the forest commitments and trapdoor information.
4. Invoke the simulator \mathcal{S}_{PC} (for univariate KZG scheme) and return $C_{\mathbf{P}}^*$.
5. Invoke the simulator \mathcal{S}_{lkp} on input the public parameters and the commitments $C_{\mathbf{P}}^*, C_{\mathcal{F}}^P$.
6. Invoke the simulator \mathcal{S}_{pred} on input the public parameters and the commitment $C_{\mathbf{P}}^*$.

Clearly, step (2) is indistinguishable from the real world. Furthermore due to the zero-knowledge properties of the PC scheme steps (1),(4) are indistinguishable from the real world. From the zero-knowledge of the sumcheck-based zkSNARK schemes, step (3) is also indistinguishable. Finally from [40, 9] we know that steps 5,6 are indistinguishable from the real world.