Scaling Intelligence: Verifiable Decision Forest Inference with *Remainder*

Modulus Labs

February 24, 2024

Abstract

We present a highly scalable instantiation of ZKML via proof of a verifiable decision forest inference circuit using a structured version of the GKR protocol [GKR15] [Tha13]. Through a combination of dataparallel GKR over a structured improvement to [ZFZS20] circuit, we are able to create GKR proofs for a decision forest of 128 trees, each of height 9, over a set of 128 inputs, each with 64 features, in under 54 seconds.

Notably, this represents a per-tree-per-sample proof time of just over 0.003s, representing a mere 180x prover-side blowup with respect to simply running the computation on CPU. In order to achieve this performance, we present several key optimizations, including a multi-stage claim aggregation optimization to the interpolation strategy presented within [Tha13], reducing the per-stage prover runtime from $O(m \cdot n \cdot 2^n)$ to $O(m \cdot (n-k) \cdot 2^n)$ for m claims over n variables, as well as a generalization of the linear-time prover trick from [XZZ+19] to the dataparallel setting, allowing us to achieve a prover time of $O(2^{s_{i+1}+b})$. We additionally provide benchmarks demonstrating the scalability of the approach, showing a sublinear relationship between proof time and adding additional trees to the forest and inputs to the batch, as well as highlighting the efficacy of both the claim aggregation optimization, i.e., a 40-60% improvement in proof generation time over the verifiable decision forest circuit, and the "Libra-Giraffe" (3.5) algorithm, i.e., a linear relationship between proof generation time and the layer size/number of dataparallel circuit copies.

Our GKR prover is combined with the Ligero polynomial commitment scheme for committing to the input layer of GKR circuits, and we call the combination *Remainder*. Our system is made fully non-interactive via Fiat-Shamir, and all benchmarks were run in a non-interactive fashion using the Poseidon [GKR$^+$19] hash function as the random oracle.

1 Introduction

Machine learning (ML) algorithms have seen drastic increases in model size (see [KSH12] to [DBK+20] [RBL+21], and [DCLT18] to [BMR+20]), accessibility ([TLI+23] [TGZ+23]), and variance of usage. In particular, larger and more capable models are being deployed in increasingly sensitive contexts, including code generation [git], education [KRGH23], and even legal [NKL+23] and medical [RIZ+17] systems.

The convergence of these trends makes it increasingly necessary, yet nearly impossible, to hold ML models *accountable*. This accountability comes in two senses – first, such models must be tested for performance, and second, the predictions from such models must be tamperproof and traceable to the original model. Currently, however, there are several key roadblocks to the above. First, ML models tend to be used in a blackbox manner and are quality controlled via extensive testing as opposed to having provable properties, and the *size* of many modern models means that only parties with large compute resources are able to run such models and generate predictions, leading to tampering risks by such parties over the model's predictions.

Moreover, large ML models often require massive amounts of data and compute to train [RSR+20], and the parameters derived from such training processes often become valuable intellectual property in and of themselves. As a result, model owners are incentivized to hide such valuable assets, further preventing, e.g., auditing or public verification techniques from working. This need for privacy-preserving ML model accountability motivates our work to improve a nascent yet rapidly evolving verifiability technique, Zero-Knowledge Machine Learning (ZKML).

1.1 ZKML

ZKML, or Zero-Knowledge Machine Learning, refers to the application of SNARKs (Succinct Non-interactive ARguments of Knowledge, originating in the seminal works [Kil92, Mic00]) to machine learning algorithms, which includes the zero-knowledge property whenever required. The vast majority of ZKML instantiations have focused on *inference*, including the one presented in this paper. ZKML as applied to machine learning inference refers to the process of generating a proof that a specific machine learning model with a fixed architecture f and weights w, when run on input x, indeed yields a specific output $\hat{y}$. Crucially, this allows for model accountability by allowing the proof recipient to attribute a result to its corresponding model and input, without necessarily revealing the model parameters themselves, satisfying the aforementioned desire of privacy-preserving accountability. Moreover, ZKML counts several key properties of SNARKs which allow it to be highly useful with respect to both scalable verifiability and model privacy:

- ZKML proofs are succinct, ranging from kilobytes to megabytes.
- ZKML proofs are easy to verify, with verification time scaling sublinearly in the size of the model.
- ZKML proofs can be verified by anybody, at any point in time.
- ZKML proofs can optionally hide any subset of the model parameters and/or the input data from the verifier.

1.2 Example Application

Consider the case of a trust-minimized price oracle, where asset appraisal prices are computed via a sophisticated ML model. In this case, the model parameters need to be kept private for IP and gamification reasons (in particular, users should *not* be able to optimize an appraisal price in either direction by modifying their inputs). This, however, exposes a large surface for potential attacks – given that the ML model operator (i.e. the party running the appraisal model and reporting results) has complete control over the algorithm's execution, they can modify the algorithm in any way they'd like to produce a desired appraisal price.

To combat these risks and ensure that the *same* appraisal model is applied uniformly to all assets, ZKML can be used – firstly, the model owner produces a public model architecture and a cryptographic commitment to the model parameters. Next, each time the model is queried for an asset appraisal price, the model operator produces both a result and an accompanying proof, verified by everyone else within the network against the aforementioned model architecture and commitment to the model parameters. Because the commitment is cryptographically hiding, it does not leak any information about the model parameters themselves. It is also cryptographically binding, i.e., every output whose accompanying proof is successfully verified using the commitment to the parameters *must* have come from running the model with the architecture using the same parameters from which the commitment was generated. In other words, the same model must have been used to generate all appraisals, and tampering would not have been possible.

Despite the potential for ZKML to meaningfully reduce tampering risks while scaling AI inference for blockchain applications, current implementations have been far from practical. Firstly, SNARKs involve large amounts of cryptographic overhead, and can take tens of thousands of times longer to generate when compared to performing the original computation alone. Additionally, SNARKs often impose huge memory loads on the prover, and can easily require terabytes of RAM to compute. On top of this, modern ML models are quite sizeable, ranging from megaFLOPs to teraFLOPs, and are themselves comprised of up to hundreds of billions of parameters [VSB+22].

The combination of these cost factors – specific to the task of verifying useful ML inference – renders the usage of generic SNARK proof systems wholly impractical (see the Cost of Intelligence paper [Lab23]). Specialization, however, offers a promising solution. We choose to specialize using a GKR-based approach to verifying ML inference.

As a concrete example, Upshot Technologies applies a very large decision forest model to the task of asset appraisals. For their first on-chain price oracle, a model composed of over 2500 gradient-boosted decision trees is used to predict granular prices for individual NFT assets. Using our implementation of a proof system for decision forest inference as detailed in section 3, we are able to verify appraisal prices produced by their models with significantly less prover overhead than previously known, as shown further in our benchmarks, specifically figures 8a, 8b.

1.3 Our Contribution

To tackle the prover-side issues of runtime and scalability with respect to model and input size in order to achieve the above, we present three primary contributions:

1. Implementing a highly-optimized version of the GKR [GKR08] proof system which we call *Remainder*, tailored primarily towards structured circuits 2.1.2, and made non-interactive via Fiat-Shamir and instantiated with the polynomial commitment from Ligero [AHIV17] over the circuit's input layer. This includes a heuristic method to aggregate multiple claims to a single one in the context of GKR and for our application that results in substantial improvements in practice.

2. Combining algorithmic advances made within Libra [XZZ+19] and Giraffe [WJB+17] for a $O(2^{b+s_{i+1}})$-time sumcheck prover (see section 3.5) over data-parallel addition and multiplication gates, where 2^b is the number of data-parallel subcircuit copies and $2^{2s_{i+1}}$ is the size of the source layer.

3. Implementing the decision forest version of the circuit from the ZKDT paper [ZFZS20] in a layered format, refining the circuit to scale with respect to inputs in a data-parallel setting, and removing the hashing bottleneck by exploiting the natural bijection between a perfect binary tree and an ordered list of length $2^k - 1$.

1.4 Related Works

We highlight related works from the ZKML literature, and defer to the Cost of Intelligence paper [Lab23] for a more comprehensive literature review. ZEN [FQZ$^+$21] was one of the earliest implementations of a ZKML system, and uses Groth16 [Gro16] as a base proof system, implementing ideas such as stranded encoding for efficient verifiable multiplication. vCNN [LKKO20] uses a combination of techniques, including Quadratic Arithmetic Programs for non-linear layers and Quadratic Polynomial Programs for convolutional layers, allowing a more efficient proving technique to be used for linear operations and a more flexible one to be used for activation functions. The work in [KHSS22] uses the flexibility of the Halo2 proof system to encode the operations of a MobileNetV2 [SHZ$^+$18], taking advantage of custom gates and lookup arguments for proving efficiency. The zkCNN paper [LXZ21] proves a VGG-16 model [SZ14] using a variant of GKR, using the sumcheck matrix multiplication algorithm from [Tha13] and introducing a new linear-time sumcheck argument to prove the computation of an FFT for efficient proving of convolutional layers within a CNN. The ZKDT paper [ZFZS20] introduces a novel circuit checking the correct inference of a decision tree, and implements the circuit in R1CS [GGPR13, SBV$^+$13, BCG$^+$13, BCR$^+$19], using Aurora [BCR$^+$19] as a backend proof system. In [CFF$^+$23] the authors introduce matrix lookup arguments, and use an encoding of a decision tree as a matrix of all possible paths within to prove correct inference execution.

Our work differs from ZEN [FQZ$^+$21], vCNN [LKKO20] and [KHSS22] in that we use a refined version of the GKR protocol as the proof system backend, and differs from zkCNN [LXZ21] in that we implement nearly all of our circuits in a data parallel, structured manner following Thaler in [Tha13] and Giraffe [WJB$^+$17] rather than relying on the techniques from Libra [XZZ$^+$19]. We differ from [CFF$^+$23] in that we encode the checking of a decision tree path as a layered circuit and prove its correctness via sumcheck rather than a matrix subset argument, and finally, from the ZKDT paper [ZFZS20] in that we encode all of the decision tree circuits as structured, layered, and data parallel circuits rather than R1CS, add an "input multiset" argument allowing for sublinear data parallel scaling in the number of input samples to process, and use a GKR backend with Ligero [AHIV17] as the polynomial commitment scheme rather than Aurora.

We present benchmarks in section 4 which demonstrate the concrete efficiency and desirable scaling properties of our implementation, as well as separate benchmarks highlighting the linear nature of our algorithm in section 3.5, which combines the arguments from Giraffe [WJB$^+$17] and Libra [XZZ$^+$19] (note that these prover techniques were only used over the subset of our circuit which could not be expressed in a structured fashion). Moreover, we run an experiment comparing proving decision forest inference against simply running the inference, and show that our "absolute" prover overheads are quite low. Concretely, they bottom out at a decision forest of 2^7 trees with 2^7 input samples, with prover overhead at 180x compared to the raw evaluation of the inputs using this model.

2 Preliminaries

In this section, we detail the necessary preliminaries for GKR and decision trees.

2.1 Preliminaries for GKR

For the preliminaries about GKR, we discuss multilinear extensions (MLE) [Tha22], the sumcheck protocol [LFKN92], GKR [GKR15], input data parallel GKR [WJB$^+$17], and GKR for structured circuits as defined in [Tha13]. By default we assume that we are working over a finite field $\mathbb{F}$.

Multilinear polynomials. A polynomial $p(x_1, \ldots, x_n)$ in n variables is multilinear if it only contains terms where the degree of every x_i is at most one in each term. We will be working with the Lagrange basis for multilinear polynomials [Tha22].

Multilinear extensions. For a functions $f : \{0,1\}^n \to \mathbb{F}$ we denote with $\tilde{f} : \mathbb{F}^n \to \mathbb{F}$ its *multilinear extension* (MLE) which is the *unique* polynomial $\tilde{f}(x_1, \ldots, x_n)$ defined as follows,

$$\tilde{f}(z) = \sum_{w \in \{0,1\}^n} f(w) \cdot \tilde{\beta}(w, z) \tag{2.1}$$

where $\widetilde{\beta}(w, z)$ is the w-th Lagrange basis polynomial that is defined as follows,

$$\widetilde{\beta}(w, z) = \prod_{i=1}^{n} (w_i \cdot z_i + (1 - w_i) \cdot (1 - z_i)).$$

Moreover, $\widetilde{\beta}(w, z)$ is the MLE of the point function (or equality predicate) that is defined as follows,

$$\beta(w, z) = \begin{cases} 1 & \text{if } w = z, \\ 0 & \text{otherwise.} \end{cases}$$

The sumcheck protocol. We use the sumcheck protocol to prove statements of the form

$$\sum_{x \in \{0,1\}^n} p(x) = y$$

where $p(x)$ is a multilinear polynomial and $y \in \mathbb{F}$. This is an interactive protocol with n rounds between a prover and a verifier who both know the polynomial $p(x)$, and is executed over a large enough finite field $\mathbb{F}$. Below we summarize how sumcheck works first for the verifier and then for the prover, without discussing its soundness and the field size requirements.

For the verifier, sumcheck works as follows:

1. In the first round, the verifier receives y and the univariate polynomial

   $$p_1(X) = \sum_{(x_2, \ldots, x_n) \in \{0,1\}^{n-1}} p(X, x_2, \ldots, x_n).$$

   It then checks that $p_1(0) + p_1(1) = y$, and if that holds, it samples $r_1 \leftarrow \mathbb{F}$ and sends r_1 to the prover.

2. In rounds $j = 2, \ldots, n-1$ the verifier receives the univariate polynomial

   $$p_j(X) = \sum_{(x_{j+1}, \ldots, x_n) \in \{0,1\}^{n-j}} p(r_1, \ldots, r_{j-1}, X, x_{j+1}, \ldots, x_n).$$

   It then checks that $p_j(0) + p_j(1) = p_{j-1}(r_{j-1})$, and if that holds, it samples $r_j \leftarrow \mathbb{F}$ and sends r_j to the prover.

3. In the final round ($j = n$), the verifier receives the univariate polynomial

$$p_n(X) = p(r_1, \ldots, r_{n-1}, X).$$

It then checks that $p_n(0) + p_n(1) = p_{n-1}(r_{n-1})$, and if that holds, it samples $r_n \leftarrow \mathbb{F}$, and finally checks that

$$p(r_1, \ldots, r_n) = p_n(r_n).$$

If the verifier has not rejected so far, it accepts.

In every sumcheck round, the verifier additionally performs a degree check on p_j. The degree bound depends on the application but is known both to the prover and the verifier. In particular, we can check this by having the prover send the evaluation of $p_j(X)$ over an a-priori known set, for example $\{0, 1, 2\}$ for a polynomial of degree two, and thus the verifier expects to receive three field elements and interpolates to obtain $p_j(X)$. This check works similarly if the prover sends $p_j(X)$ in the coefficient form.

For the prover, sumcheck works as follows:

1. In the first round the prover sends y and the univariate polynomial

$$p_1(X) = \sum_{(x_2,\ldots,x_n)\in\{0,1\}^{n-1}} p(X, x_2, \ldots, x_n).$$

2. In rounds, $j = 2, \ldots, n-1$ the prover sends the univariate polynomial

$$p_j(X) = \sum_{(x_{j+1},\ldots,x_n)\in\{0,1\}^{n-j}} p(r_1, \ldots, r_{j-1}, X, x_{j+1}, \ldots, x_n).$$

Note that before round j the prover has received $(r_1, \ldots, r_{j-1})$ from the verifier, and thus is able compute $p_j(X)$.

3. In the final round ($j = n$), the prover sends the univariate polynomial

$$p_n(X) = p(r_1, \ldots, r_{n-1}, X).$$

The GKR protocol. The GKR protocol that we summarize below, was introduced in [GKR08] to prove claims for circuits that can be expressed in layers, such that each layer is defined using values only from the previous one. In particular, let C be a circuit that can be expressed using $(d+1)$ layers $(L_d, \ldots, L_0)$ with L_0 being the output layer and L_d being the input layer.

For an input x, circuit C, and a claim that $C(x) = y$, the GKR protocol uses the sumcheck protocol to reduce the soundness of the output layer L_0 to layer L_1 (i.e. the previous layer), continues in a chain-argument manner to reduce the soundness of L_1 to L_2, all the way until we reduce soundness for L_{d-1} to that of L_d. The soundness of L_d at the end of the sumcheck protocol can be checked directly by the verifier for the given input x.

In more detail, we represent the 2^{s_i} values of the i-th layer using a vector of field values that is indexed (labeled) using s_i bits, $V_i(p_1) : \{0,1\}^{s_i} \to \mathbb{F}$. These values are typically the output of addition and multiplication gates that take as input values from the previous layer (which is $V_{i+1}(p_1)$ and contains $2^{s_{i+1}}$ values). This is expressed as follows,

$$V_i(p_1) = \sum_{(x,y)\in\{0,1\}^{2s_{i+1}}} \mathsf{add}_i(p_1, x, y)(V_{i+1}(x) + V_{i+1}(y)) + \mathsf{mult}_i(p_1, x, y)(V_{i+1}(x) \cdot V_{i+1}(y)) \quad (2.2)$$

In the above expression, the $\mathsf{add}_i(p_1, x, y)$ and $\mathsf{mult}_i(p_1, x, y)$ are predicates mapping $\{0,1\}^{s_i+2s_{i+1}} \to \{0,1\}$, by outputting 1 if there is a addition gate (resp. multiplication) with output label p_1 (at layer i) and input labels x, y (at layer $i+1$), and 0 otherwise. Note that for every layer we pad the number of values to the nearest power-of-two.

For adjacent layers i and $i+1$, using the MLEs for the terms in Equation 2.2, the sumcheck protocol is used to prove the following claim,

$$\widetilde{V}_i(g) = \sum_{(x,y)\in\{0,1\}^{2s_{i+1}}} \widetilde{\mathsf{add}}_i(g,x,y)(\widetilde{V}_{i+1}(x) + \widetilde{V}_{i+1}(y)) + \widetilde{\mathsf{mult}}_i(g,x,y)(\widetilde{V}_{i+1}(x)\cdot\widetilde{V}_{i+1}(y)) \tag{2.3}$$

where the value $g \in \mathbb{F}^{s_i}$ is fixed (at random) from the claim of the previous layer.

Here, we note three important points. First, the verifier must evaluate on its own the MLEs $\widetilde{\mathsf{add}}_i(\cdot,\cdot,\cdot), \widetilde{\mathsf{mult}}_i(\cdot,\cdot,\cdot)$ as these are needed during the sumcheck protocol, including all the random challenges that arise during the sumcheck protocol. This does not increase the round complexity.

Second, at the end of the GKR protocol, when reducing a claim from layer L_{d-1} to the input layer L_d, the verifier must evaluate on its own $\widetilde{V}_d(z)$ at a random point $r_d \in \mathbb{F}^{s_d}$. Third, at the end of each GKR round, the sumcheck protocol for Equation 2.3 reduces the soundness of claim $\widetilde{V}_i(r_i)$ to two claims $\widetilde{V}_{i+1}(r_{i+1}), \widetilde{V}_{i+1}(r'_{i+1})$ for random r_i, r_{i+1}, r'_{i+1}. To avoid an exponential blow-up in the number of claims as we progress towards the input layer, there are standard ways to combine the two claims $\widetilde{V}_{i+1}(r_{i+1}), \widetilde{V}_{i+1}(r'_{i+1})$ into one claim [GKR08, Tha13] for Equation 2.3. We discuss in detail our specialized approach to claim aggregation in Section 3.4.

2.1.1 Data parallel circuits for GKR

The use of GKR for a circuit C but across $m = 2^{s_m}$ different inputs $\{x_1, \dots, x_m\}$, can be optimized compared to running m independent executions of the GKR protocol for each x_j. This is done by re-writing Equation 2.3 to include a label for the j-th instance $C(x_j)$, as follows:

$$\begin{aligned}\widetilde{V}_i(g_2, g_1) = \sum_{(p_2,p_1,x,y)\in\{0,1\}^{b+s_i+2s_{i+1}}} & \widetilde{\beta}(g_2,p_2)\cdot\widetilde{\beta}(g_1,p_1)\cdot \\ & \Big(\widetilde{\mathsf{add}}_i(p_1,x,y)\cdot\big(\widetilde{V}_{i+1}(p_2,x)+\widetilde{V}_{i+1}(p_2,y)\big)+ \\ & \widetilde{\mathsf{mult}}_i(p_1,x,y)\cdot\big(\widetilde{V}_{i+1}(p_2,x)\cdot\widetilde{V}_{i+1}(p_2,y)\big)\Big)\end{aligned}$$

Note that we now have $\widetilde{V}_i : \mathbb{F}^m \times \mathbb{F}^{s_i} \mapsto \mathbb{F}$, rather than $\widetilde{V}_i : \mathbb{F}^{s_i} \mapsto \mathbb{F}$. In particular, for binary $\ell \in \{0,1\}^m, z \in \{0,1\}^{s_i}$, $\widetilde{V}_i(\ell, z)$ is the value of the zth output within the ℓth circuit copy. In this case, the overall circuit depth is similar to that of C and its width is implied by the parallel stacking of the same circuit C for m times because of the execution of C on m different inputs. While such a "circuit stacking" could model parallel computation, the final result is still a *single* circuit C' on which the GKR protocol is used. The repetitive structure of C' allows for certain advantages when compared to m independent executions of C (even if these were executed concurrently), such as smaller proof size, and prover and verifier complexity [WJB+17]. We refer to [Tha22, ch. 4] and [Tha13] for more information on regular circuits (see also Section 2.1.2) and data parallel circuits, as well as a more thorough introduction to the GKR protocol.

2.1.2 Structured circuits for GKR

The canonic generalized layerwise circuit relationship described above for GKR in Equation 2.3 is extremely flexible and yields a highly parallelizable, linear-time prover as shown in [XZZ+19].

However, it comes with two major downsides, one for the prover and verifier and the other for the verifier alone. For the following, assume that $x \in \{0,1\}^{s_{i+1}}$ is bound to $u \in \mathbb{F}^{s_{i+1}}$, and that $y \in \{0,1\}^{s_{i+1}}$ is bound to $v \in \mathbb{F}^{s_{i+1}}$.

The first major downside is that the verifier must do work *linear* in the size of the overall circuit. This is because without any form of structure within the circuit, the best algorithm for evaluating $\widetilde{\mathsf{add}}_i(g,u,v)$ and $\widetilde{\mathsf{mult}}_i(g,u,v)$ for $g \in \mathbb{F}^{s_i}$, $u, v \in \mathbb{F}^{s_{i+1}}$ is linear in the total number of addition and multiplication gates, respectively. In other words,

$$\widetilde{\mathsf{add}}_i(g,u,v) = \sum_{(p_1,x,y) \in \mathcal{N}_{\mathrm{add}_i}} \widetilde{\beta}(g,p_1) \cdot \widetilde{\beta}(u,x) \cdot \widetilde{\beta}(v,y) \tag{2.4}$$

where $\mathcal{N}_{\mathrm{add}_i} = \{(p_1,x,y) \mid \mathsf{add}_i(p_1,x,y) = 1\}$.

A similar relationship holds for $\widetilde{\mathsf{mult}}_i(g,u,v)$. Indeed, implementing the above equality naively yields an $O(C\log(C))$ verifier runtime, as each $\widetilde{\beta}$ term requires s_i or s_{i+1} time to compute – rather, the precomputation step described within [XZZ+19] is what allows the verifier to achieve $O(C)$ for the above.

The second major downside is that the claims which are generated at the end of sumcheck over layer i on layer $i+1$ in the canonic relationship are in the form $\widetilde{V}_{i+1}(u) = h_1, \widetilde{V}_{i+1}(v) = h_2$ for some prover-claimed values $h_1, h_2 \in \mathbb{F}$. Notably, u and v are completely independent of one another, and thus the claim aggregation strategy described in [Tha13] has a prover runtime of $O(2 \cdot s_{i+1} \cdot 2^{s_{i+1}})$, as described in much greater detail in Section 3.4, and a verifier runtime of $O(2s_{i+1})$ (note that the constant 2 actually describes the number of claims – in general, with m claims we'd see a prover runtime of $O(m \cdot s_{i+1} \cdot 2^{s_{i+1}})$ and a verifier runtime of $O(m \cdot s_{i+1})$).

The key insight from [Tha13], then, is that a large class of circuits are *structured*, i.e. their wiring predicates (which we would write as $\widetilde{\mathsf{add}}_i$ and $\widetilde{\mathsf{mult}}_i$) are predictable/repetitive, and can be evaluated in logarithmic time by the verifier – this allows the verifier in particular to achieve an $O(d\log C)$ overall runtime for GKR verification (not including input layer claim verification), and additionally produces claims whose evaluation points match in many coordinates, e.g. $\widetilde{V}_{i+1}(0, r_2, \ldots, r_{s_{i+1}}) = h_1$ and $\widetilde{V}_{i+1}(1, r_2, \ldots, r_{s_{i+1}}) = h_2$, giving us the basis for our claim aggregation optimization for both prover and verifier in Section 3.4. Formally, the wiring predicates within the circuit must be *regular*, and must be *similar* to each other (we defer to [Tha13], Theorem 1, for definitions and a detailed description of such circuits); for the circuits described in this paper, we simply impose the sufficient (but not entirely necessary) condition that the binary indices used on the LHS of every layerwise relationship are the only ones used on the RHS, e.g.

$$V_i(b_1, \ldots, b_n) = V_{i+1}(0, b_1, \ldots, b_n) \cdot V_{i+1}(1, b_1, \ldots, b_n) \tag{2.5}$$

in the case of a binary multiplication tree, multiplying the first and second halves of the previous layer's data element-wise. The equivalent MLE relationship is the following:

$$\widetilde{V}_i(g_1, ...g_n) = \sum_{b_1,\ldots,b_n \in \{0,1\}^n} \widetilde{\beta}(g_1, ..., g_n, b_1, ..., b_n) \cdot \left(\widetilde{V}_{i+1}(0, b_1, ..., b_n) \cdot \widetilde{V}_{i+1}(1, b_1, ..., b_n)\right) \tag{2.6}$$

The claims generated by sumcheck (assuming $b_i \in \{0,1\}$ is bound to $r_i \in \mathbb{F}$) are exactly of the form $\widetilde{V}_{i+1}(0, r_1, ..., r_n) = h_1$ and $\widetilde{V}_{i+1}(1, r_1, ..., r_n) = h_2$, allowing for an $O((n-k) \cdot m \cdot 2^n)$ prover-time claim reduction strategy, where k is the number of indices where all of the challenges are the same (again, see 3.4 for more details).

2.2 Preliminaries for decision trees

In this subsection, we describe decision trees and decision forests. In particular, a binary decision tree is akin to a branching version of a game of "20 questions", where every question's answer corresponds with traversing either the left or right branch of that particular node. The leaves of a decision tree are associated with a prediction value, and for any input, traversing the tree using the input's features and asking each node's "associated question" yields the prediction of the tree on that particular input. In the case of "regression" decision trees, where predictions are real-valued, a decision *forest* is simply a set of decision trees whose prediction function over a given input is computed by simply adding the individual prediction values from each decision tree. Below, we formalize the above intuition and describe the prediction (inference) function of decision trees in detail.

Notation. We use $[m]$ to denote the set of positive integers $\{1, \ldots, m\}$.

2.2.1 Decision Forests

We consider a forest consisting of m binary decision trees $\{T_j\}_{j \in [m]}$. We assume that each tree is "perfect", i.e. that any path from the root to a leaf has the same length[1]. The number of nodes on such a path is the "height" (or "depth") of the tree; we assume that all trees have the same height h.

Every tree T_j is defined as follows: every internal (i.e. non-leaf) tree node contains a tuple: `(id, feature_index, threshold)`. The tree leaf nodes contain a tuple `(id, prediction)`. In both cases, `id` is the index of the node in the tree, in a left-to-right and layer-wise ordering. The `id` numbering of each tree starts from 0, at the root node.

An input to the decision tree is a vector x, consisting of pairs of the form `(feature_index, value)` where `feature_index` ranges over $[k_{\mathsf{fi}} - 1]$ for some fixed k_{fi}. Except for `prediction`, which is a *signed* value, all other values in T_j and x are *non-negative*, and all these values are embedded appropriately in a large enough finite field $\mathbb{F}$.

In particular, let the tree T_j be defined as:

$$T_j = \left[\overbrace{\left(0, \mathsf{fi}_0^{(j)}, \mathsf{thr}_0^{(j)}\right), \ldots, \left(2^{(h-1)} - 2, \mathsf{fi}_{2^{(h-1)}-2}^{(j)}, \mathsf{thr}_{2^{(h-1)}-2}^{(j)}\right)}^{\text{internal nodes}}, \overbrace{\left(2^{(h-1)} - 1, \mathsf{pred}_{2^{(h-1)}-1}^{(j)}\right), \ldots, \left(2^h - 2, \mathsf{pred}_{2^h-2}^{(j)}\right)}^{\text{leaf nodes}} \right]$$

Where we write fi, thr, pred as short-hand for feature index, threshold and prediction, respectively. The tree T_j has $2^{(h-1)} - 1$ internal nodes, $2^{(h-1)}$ leaf nodes, for a total of $2^h - 1$ nodes. In our case, all root to leaf paths are have the same length, since our trees are perfect.

The decision forest prediction for an input x under $\{T_j\}_{j \in [m]}$ is simply the sum of each individual tree's prediction over x, i.e. $\sum_{j=1}^{m} T_j(x)$. For every j, the prediction $T_j(x)$ is computed as follows. First, let x be

$$x = \left[\left(\mathsf{fi}_0^{(x)}, v_0^{(x)}\right), \ldots, \left(\mathsf{fi}_{k_{\mathsf{fi}}-1}^{(x)}, v_{k_{\mathsf{fi}}-1}^{(x)}\right) \right]$$

To compute $T_j(x)$, we first select the pair $\left(\mathsf{fi}_t^{(x)}, v_t^{(x)}\right)$ from x such that $\mathsf{fi}_0^{(i)} = \mathsf{fi}_t^{(x)}$, i.e. select the tuple from x with a `feature_index` that matches that of the root node in T_j. If $v_t^{(x)} \geq \mathsf{thr}_0^{(j)}$ we move on to the right child node of the root node, otherwise we move on to the left child. This process is repeated until a leaf node is reached; the prediction of this leaf node is the prediction of the tree. At a high level, we have:

[1] Many decision trees are not perfect. We preprocess such trees by padding them with dummy nodes that do not affect tree predictions.

```
Algorithm 1: Compute $T_j(x)$
    Input: $T_j, h, x$
    Result: $\mathsf{pred}^{(j)}_{\mathsf{id}}$ for some $\mathsf{id} \in \{2^{(h-1)} - 1, \ldots, 2^h - 2\}$
    // init to the root node id
 1  $\mathsf{id} = 0$ ;
 2  for $\mathsf{height} = 1, \ldots, h - 1$ do
 3      Select $t$ such that $\mathsf{fi}^{(j)}_{\mathsf{id}} = \mathsf{fi}^{(x)}_t$ // Such a $t$ always exists.  Assume Select exists.    ;
 4      if $v^{(x)}_t \geq \mathsf{thr}^{(j)}_{\mathsf{id}}$ then
            // right child
 5          $\mathsf{id} = 2 \cdot \mathsf{id} + 2$;
 6      else
            // left child
 7          $\mathsf{id} = 2 \cdot \mathsf{id} + 1$;
 8      end
 9  end
10  return $\mathsf{pred}^{(j)}_{\mathsf{id}}$;
```

Note that for the prediction $T_j(x)$ to be well-defined, the input x must contain information for every `feature_index` that could appear in any tree T_j. This is modeled by having a universe of feature indices (i.e. a fixed set) which is used for every T_j, and then having every x to contain a pair $(\mathsf{fi}_{t'}, v_{t'})$ for every $\mathsf{fi}_{t'}$ in that universe. Thus, the `Select` procedure in Algorithm 1 at Line 3 is well-defined. Finally, in our proof system we prove the verification of the inference given by Algorithm 1, and thus the `Select` procedure is not required in our case. This is made clear in Section 3.3.

3 Decision forest circuit using the *Remainder* proof system

Remainder is our implementation of a non-interactive, data parallel GKR proof system with Ligero as the polynomial commitment. In implementing *Remainder*, we adopt techniques from [GKR15, Tha13, WJB+17, AHIV17, XZZ+19, ZFZS20, DP23], and incorporate several of our own improvements and optimizations, notably those described below in Sections 3.4 and 3.5.

Using *Remainder*, we circuitize a decision forest model which we describe in more detail throughout the rest of this section. In this manuscript, we omit the details of our recursion strategy that results in a zkSNARK proof system, and instead focus on the GKR prover, yielding a SNARK proof system with sublinear proof size and verification time due to a) the use of the Ligero polynomial commitment scheme for the GKR prover input, and b) the use of structured GKR circuits as described earlier in Section 2.1.2 and in [Tha13]. For our benchmarks we use Poseidon [GKR+21] as the hash function of choice for ease of recursion: we use it to generated the random challenges in the Fiat-Shamir transform and as a collision-resistant hash function to hash columns and subsequently create a Merkle tree out of the column hashes in the context of Ligero. Our experiments were performed over the BN-254 scalar field, and that further concrete improvements might be seen with either a smaller field and/or a different hash function.

3.1 Verifiable Decision Forest circuit overview

For a set of n inputs $S_x = \{x_1, \ldots, x_n\}$ given as input to a decision forest consisting of m decision trees $S_T = \{T_j\}_{j \in [m]}$, we create a single data parallel GKR circuit (see Section 2.1.1 and [Tha22, ch. 4] for more details) over mn different (tree, input) pairs, where each data parallel instance handles the prediction of a single decision tree T_j for a single input x_i. The output of such a circuit is thus mn different values, with the overall decision forest prediction for each sample x_i computed via a simple sum over all the trees: the prediction pred_{S_T,x_i} for the input x_i is $\sum_{j=1}^{m} T_j(x_i) = \sum_{j=1}^{m} \mathsf{pred}_i^{(j)} = \mathsf{pred}_{S_T,x_i}$.

There are two main parts in our data parallel GKR proof system. Firstly, we elaborate on the contents of the entire input to the GKR prover, i.e. a decision tree T, an input x for T, auxiliary information aux that is essentially the witness for computing $T(x) = y$, and a tuple of field elements for a specific purpose that will be made clear later. Secondly, we describe the data parallel GKR circuit that uses this entire input to verify that $T(x) = y$.

In our case, the GKR prover uses a commitment to the tree T, a commitment to input x, and a commitment to aux. The x used is the result of a preprocessing step that makes it convenient to work with our GKR circuit.

3.2 Data parallel GKR input

In this section, we describe the input $\mathbf{x}_{\mathsf{GKR}}$ used by our circuit to prove decision forest inferences, which we describe next in Section 3.3.

For a fixed set $S_T = \{T_j\}_{j\in[m]}$ of decision trees, the data parallel GKR input $\mathbf{x}_{\mathsf{GKR}}$ includes the following: a tuple of random field elements $(z_{\mathsf{msi}}, z_{\mathsf{fi}}, z_{\mathsf{tp}}, r_{\mathsf{msi}}, r_{\mathsf{msi}})$ (see below), a set of predictions $\mathsf{Pred}_{(S_T,S_x)} = \{T_0(x_0) = \mathsf{pred}_{T_0,x_0}, \ldots, T_0(x_n) = \mathsf{pred}_{T_0,x_n}, \ldots, T_m(x_n) = \mathsf{pred}_{T_m,x_n}\}$, and finally the following values which are contained in multilinear extensions that are evaluated by the verifier using the Ligero commitment via commit-and-open proofs:

1. A commitment to the decision forest S_T: $\mathsf{com}_{S_T} = \mathsf{LigeroCommit}(S_T)$. Note that the same commitment for S_T is used each time we run our proof system with a different input set S_x (defined below) unless any of the trees within S_T needs to be updated.

2. A commitment to n inputs $S_x = \{x_1, \ldots, x_n\}$: $\mathsf{com}_{S_x} = \mathsf{LigeroCommit}(x_1, \ldots, x_n)$. This is used across all data parallel GKR instances.

3. A commitment to mn units of auxiliary information, over all inputs $\{x_1, \ldots, x_n\}$ and all trees $\{T_1, \ldots, T_m\}$:

$$\begin{aligned}\mathsf{com}_{(S_T,S_x)} = \mathsf{LigeroCommit}\big(&\{\mathsf{aux}_{(T_j,\bar{x}_i)} \mid 0 \le i < n, 0 \le j < m\} \cup \\ &\{\mathsf{aux}_{(S_T,\bar{x}_i)} \mid 0 \le i < n\} \cup \\ &\{\mathsf{aux}_{(T_j,S_x)} \mid 0 \le j < m\}\big)\end{aligned}$$

   Notice that here we use $\bar{x}_i$ instead of x_i because it is convenient for a tree T_j to only use the subset of attributes from the input x_i which are checked in the path induced while computing $T_j(x_i)$.

The tuple of random field elements $(z_{\mathsf{msi}}, z_{\mathsf{fi}}, z_{\mathsf{tp}}, r_{\mathsf{msi}}, r_{\mathsf{msi}})$ is send beforehand by the verifier in the interactive case, and is sampled accordingly in the non-interactive case via Fiat-Shamir. These field elements are used in Section 3.3.

Below we describe in more detail the committed values in the above Ligero commitments.

The decision forest S_T. We instead describe the committed values which are present for *each* decision tree $T \in S_T$, and note that the committed values for S_T are a simple concatenation of those values. First, recall that the decision tree T which is defined in a root to leaf order as in Section 2.2.1 is as follows:

$$T = \left[\overbrace{(0, \mathsf{fi}_0, \mathsf{thr}_0), \ldots, \left(2^{(h-1)} - 2, \mathsf{fi}_{2^{(h-1)}-2}, \mathsf{thr}_{2^{(h-1)}-2}\right)}^{\text{internal nodes}}, \overbrace{\left(2^{(h-1)} - 1, \mathsf{pred}_{2^{(h-1)}-1}\right), \ldots, \left(2^h - 2, \mathsf{pred}_{2^h-2}\right)}^{\text{leaf nodes}} \right]$$

The input set $\{x_1, \ldots, x_n\}$. Each x_i contains k pairs as defined in Section 2.2.1

$$x_i = \left[\left(\mathsf{fi}_{i_0}^{(x_i)}, v_{i_0}^{(x_i)}\right), \ldots, \left(\mathsf{fi}_{i_{|x_i|-1}}^{(x_i)}, v_{i_{|x_i|-1}}^{(x_i)}\right) \right]$$

Note that n is padded to the nearest power-of-two and all x_i have the same length k_{fi}.

The auxiliary information $\mathsf{aux}_{(T_j,x_i)}, \mathsf{aux}_{(S_T,x_i)}, \mathsf{aux}_{(T_j,S_x)}$. Note that the first auxiliary information listed is present for *each* tree T_j and input x_i, while the second auxiliary information is present only for each input x_i, and the third auxiliary information is present only for each tree T_j. We describe each of the aforementioned auxiliary information in detail below. Let h be the height of the decision tree T_j, and in our case h is the same for all trees T_j. In detail, $\mathsf{aux}_{(T_j,x_i)}$ contains the following:

- A vector $\bar{x}_{i,j}$ of $(h-1)$ pairs,

  $$\bar{x}_{i,j} = \left[\left(\overline{\mathsf{fi}}_0, \bar{v}_0\right), \ldots, \left(\overline{\mathsf{fi}}_{h-2}, \bar{v}_{h-2}\right) \right]$$

  where each pair is given in the order that is used during the prediction $T_j(x_i)$, i.e. in the induced path of $T_j(x_i)$ the k-th decision node has feature id $\overline{\mathsf{fi}}_k^{(i,j)}$.

- The prediction output: $T_j(\bar{x}_{i,j}) = T_j(x_i) = \mathsf{pred}_{T_j,x_i}$

- A vector of $h-1$ internal tree nodes and a leaf node, for a total of h tree nodes which defines the path induced by $T_j(\bar{x}_{i,j})$:

  $$\mathsf{path}_{\bar{x}_{i,j}} = [(\overline{\mathsf{id}}_0, \overline{\mathsf{fi}}_0, \overline{\mathsf{thr}}_0), (\overline{\mathsf{id}}_1, \overline{\mathsf{fi}}_1, \overline{\mathsf{thr}}_1), \ldots, (\overline{\mathsf{id}}_{h-2}, \overline{\mathsf{fi}}_{h-2}, \overline{\mathsf{thr}}_{h-2}), (\overline{\mathsf{id}}_{h-1}, \overline{\mathsf{pred}}_{T_j,x_i})]$$

- The *signed* binary decomposition of $\left(\overline{\mathsf{thr}}_k - v_k^{(\bar{x})}\right)$, for $k = 0, \ldots, h-2$, defined as

  $$\mathsf{SgnBitDecomp}\left(\overline{\mathsf{thr}}_k - v_k^{(\bar{x})}\right) = \left(b_{(k,0)}^{\bar{x}}, \ldots, b_{(k,\ell-1)}^{\bar{x}}, b_{(k,\mathsf{sgn})}^{\bar{x}}\right)$$

  where $\ell = \left\lceil \log_2 \left(\left| \overline{\mathsf{thr}}_k - v_k^{(\bar{x})} \right| \right) \right\rceil$ and $b_{(k,\mathsf{sgn})}^{\bar{x}}, b_{(k,s)}^{\bar{x}} \in \{0,1\}$ with

  $$\left(\overline{\mathsf{thr}}_k - v_k^{(\bar{x})}\right) = (-1)^{b_{(k,\mathsf{sgn})}^{\bar{x}}} \cdot \left(\sum_{s=0}^{\ell-1} 2^s \cdot b_{(k,s)}^{\bar{x}} \right)$$

  That is, the bits $b_{(k,s)}^{\bar{x}}$ represent the binary decomposition of the absolute value $\left| \overline{\mathsf{thr}}_k - v_k^{(\bar{x})} \right|$.

Next, we describe the contents of $\mathsf{aux}_{(S_T,\bar{x}_i)}$ which is the auxiliary information provided per input $\bar{x}_i$ but is aggregated across all trees in S_T as we explain below.

- The *unsigned* binary decomposition of ct^i_k which is a counter for the number of times the k-th pair of x_i appears in total in any of the $\bar{x}_{i,j}$, for $j = 0, \ldots, (h-2)$, even if this counter is 0. That is,

$$\mathsf{BitDecomp}(\mathsf{ct}^i_k) = \left(\mathsf{ct}^i_{(k,0)}, \ldots, \mathsf{ct}^i_{(k,\ell-1)}\right)$$

where $\ell = \lceil \log_2 \left(\mathsf{ct}^i_k\right) \rceil$ and $\mathsf{ct}^i_{(k,s)} \in \{0,1\}$ with $\mathsf{ct}^i_k = \sum_{s=0}^{\ell-1} 2^s \cdot \mathsf{ct}^i_{(k,s)}$

Finally, we describe the contents of $\mathsf{aux}_{(T_j,S_x)}$which is the auxiliary information provided per tree T_j but is aggregated across all inputs in S_x as we explain below.

- The *unsigned* binary decomposition of ct^j_k, which is a counter for the number of times the k-th node of T_j was accessed in total by the paths induced by the predictions $T_j(x_1), \ldots, T_j(x_n)$, even if this counter is 0. That is,

$$\mathsf{BitDecomp}(\mathsf{ct}^j_k) = \left(\mathsf{ct}^j_{(k,0)}, \ldots, \mathsf{ct}^j_{(k,\ell-1)}\right)$$

where $\ell = \left\lceil \log_2(\mathsf{ct}^j_k) \right\rceil$ and $\mathsf{ct}^j_{(k,s)} \in \{0,1\}$ with $\mathsf{ct}_j = \sum_{s=0}^{\ell-1} 2^s \cdot \mathsf{ct}^j_{(k,s)}$.

3.3 Data parallel GKR circuit for decision forests

In this section, we describe the circuit for the prediction $T_j(x_i) = \mathsf{pred}_{i,j}$ that takes as input $\mathbf{x}_{\mathsf{GKR}}$ defined in Section 3.2. There are three main components to our circuit:

- The *input multiset* circuit, which proves that, for a given sample x_i and across all trees T_j, that the reordered samples $\bar{x}_{ij}$ consistent with the paths taken in each tree are composed of only features present in the original input x_i.
- The *path validity* circuit, which, given a reordered sample $\bar{x}_{ij}$ and a path, $\mathsf{path}_{\bar{x}_{i,j}}$, checks that the decisions implicit within $\mathsf{path}_{\bar{x}_{i,j}}$ are consistent with the features in $\bar{x}_{ij}$.
- The *tree multiset* circuit, which given a set of paths $\mathsf{path}_{\bar{x}_{i,j}}$ and the tree nodes within a tree T_j proves that the nodes within all of the paths are also present within the tree.

The combination of these three circuits is sufficient to prove the correct inference of a decision forest model $\{T_j\}$ over a set of inputs $\{x_i\}$ (see Theorem 4.2 in [ZFZS20]).

3.3.1 Input Multiset Circuit

We prove that $\{\bar{x}_{i,j}\}_{j=0}^{m-1}$ is a multiset of x_i (note that x_i is a set of elements itself), for every x_i in the input set $S_x = \{x_1, \ldots, x_n\}$.

First, we construct a circuit that uses a random field element z_{msi} (defined in Section 3.2) to combine (or "pack"), each tuple in x_i and $\bar{x}_i$ into one field element, respectively, as follows:

$$x_i^{(p)} = \left[\left(\mathsf{fi}_0^{(x_i)} + v_0^{(x_i)} \cdot z_{\mathsf{msi}}\right), \ldots, \left(\mathsf{fi}_{|x_i|-1}^{(x_i)} + v_{|x_i|-1}^{(x_i)} \cdot z_{\mathsf{msi}}\right)\right]$$

$$\bar{x}_i^{(p)} = \bigcup_{j=0}^{m-1} \left[\left(\overline{\mathsf{fi}}_0^{(\bar{x}_{i,j})} + \bar{v}_0^{(\bar{x}_{i,j})} \cdot z_{\mathsf{msi}}\right), \ldots, \left(\overline{\mathsf{fi}}_{(h-2)}^{(\bar{x}_{i,j})} + \bar{v}_{(h-2)}^{(\bar{x}_{i,j})} \cdot z_{\mathsf{msi}}\right)\right]$$

where $x_i^{(p)}$ and $\bar{x}_i^{(p)}$ denote the "packed" version of x_i and $\bar{x}_i$ with $\bar{x}_i$ aggregated over all trees, respectively.

Finally, using a random field element r_{msi} (defined in Section 3.2) our circuit computes the following difference:

$$\prod_{k=0}^{|x_i|-1} \left(r_{\mathsf{msi}} - x_i^{(p)}[k] \right)^{\mathsf{ct}_k^i} - \prod_{j=0}^{m-1} \prod_{k=0}^{h-2} \left(r_{\mathsf{msi}} - \bar{x}_{i,j}^{(p)}[k] \right) \tag{3.1}$$

where $x_{i,j}^{(p)}[k] = \left(\mathsf{fi}_k^{(x_i)} + v_k^{(x_i)} \cdot z_{\mathsf{msi}} \right)$ for all $k \in \{0, \ldots, |x_i|-1\}$, and $\bar{x}_{i,j}^{(p)}[k]$ is defined analogously, for all $k \in \{0, \ldots, h-2\}$. Note that $|x_i|$ is the same for all i in our case.

By the Schwartz-Zippel lemma we have that

- If $\bar{x}_i$ is a multiset of x_i, then the difference 3.1 will always be 0.
- If $\bar{x}_i$ is *not* a multiset of x_i, then the difference 3.1 will be nonzero with high probability.

In fact, this is the (pre-defined) circuit output that the GKR verifier expects for this assuming an honest prover, i.e. the zero output. This (difference) amounts essentially to the low-degree polynomials $p(z) = \prod_{k=0}^{|x_i|-1} \left(z - x_i^{(p)}[k] \right)$ and $\bar{p}(z) = \prod_{k=0}^{h-2} \left(z - \bar{x}_{i,j}^{(p)}[k] \right)$ being equal, since the product of the monomials of their roots results in the exact same polynomial over $\mathbb{F}$, i.e. $p(z) = \bar{p}(z)$.

3.3.2 Path Validity Circuit

We prove that $\mathsf{path}_{\bar{x}}$ is a *valid* path for a decision tree, based on the node `id` values given in $\mathsf{path}_{\bar{x}}$. In addition, we prove that it is a *consistent* prediction path. Below we show this for a fixed $\bar{x}$ and a fixed T_j since the process is the same for every $(\bar{x}_i, T_j)$ pair. For this step, the circuits will be using $\mathsf{path}_{\bar{x}}$, $\bar{x}$ and $\mathsf{aux}_{(T,\bar{x})}$, as listed below:

$$\mathsf{path}_{\bar{x}} = [(\overline{\mathsf{id}}_0, \overline{\mathsf{fi}}_0, \overline{\mathsf{thr}}_0), (\overline{\mathsf{id}}_1, \overline{\mathsf{fi}}_1, \overline{\mathsf{thr}}_1), \ldots, (\overline{\mathsf{id}}_{h-2}, \overline{\mathsf{fi}}_{h-2}, \overline{\mathsf{thr}}_{h-2}), (\overline{\mathsf{id}}_{h-1}, \overline{\mathsf{pred}}_{T,x})]$$

The *signed* binary decomposition of $\left(\overline{\mathsf{thr}}_k - v_k^{(\bar{x})} \right)$, for $k = 0, \ldots, (h-2)$, that is,

$$\mathsf{SgnBitDecomp}\left(\overline{\mathsf{thr}}_k - v_k^{(\bar{x})} \right) = \left(b_{(k,0)}^{\bar{x}}, \ldots, b_{(k,\ell-1)}^{\bar{x}}, b_{(k,\mathsf{sgn})}^{\bar{x}} \right)$$

Path Validity Circuit

To prove that $\mathsf{path}_{\bar{x}}$ is a *valid* path within decision tree T, i.e., it contains a valid parent and child node `id` sequence within T, we construct a circuit that verifies the selection of left or right child node `id` based on $b_{(k,\mathsf{sgn})}^{\bar{x}}$. Assume the current tree node is `id`. If $b_{(k,\mathsf{sgn})}^{\bar{x}} = 1$ (i.e. the difference $\overline{\mathsf{thr}}_k - v_k^{(\bar{x})}$ is negative) then we select the left child node with an ID of $(2 \cdot \mathtt{id} + 1)$; otherwise, we select the right child node with an ID of $(2 \cdot \mathtt{id} + 2)$. Moreover, we use a multiset argument to show that every node within $\mathsf{path}_{\bar{x}}$ is also a node within T.

Path Consistency Circuit

To prove that $\mathsf{path}_{\bar{x}}$ is a *consistent* prediction path, we construct a circuit that first verifies the following, i.e. checks both that the difference computed in circuit by the prover is within a particular range, *and* that the sign bit $b_{(k,\mathsf{sgn})}^{\bar{x}}$ correctly matches the sign of the difference:

$$\left(\overline{\mathsf{thr}}_k - v_k^{(\bar{x})} \right) = (-1)^{b_{(k,\mathsf{sgn})}^{\bar{x}}} \cdot \sum_{s=0}^{\ell-1} 2^s \cdot b_{(k,s)}^{\bar{x}}$$

next, we show that the attributes accessed in each of the nodes within $\mathsf{path}_{\bar{x}}$ match exactly those in $\bar{x}$:

$$\mathsf{path}_{\bar{x}}.\overline{\mathsf{fi}}_k = \bar{x}.\overline{\mathsf{fi}}_k^{(\bar{x})}$$

Here, we do not check that $\overline{\mathsf{id}}_0 = 0$ or that $\mathsf{path}_{\bar{x}}$ belongs in T. This is handled in the next step. Finally, for $\mathsf{pred}_{T,x} \in \mathsf{Pred}_{(T,S_x)}$, our circuit computes,

$$\mathsf{path}_{\bar{x}}.\overline{\mathsf{pred}}_{T,x} - \mathsf{pred}_{T,x}$$

which is expected to output zero if the correct prediction was given as part of the input. This last step, forces the output of our circuit to be the all zero vector.

3.3.3 Tree Multiset Check

We prove for each tree T_j that the nodes within $\cup_i \mathsf{path}_{\bar{x}_{i,j}}$ are all valid nodes within T_j.

First, we construct a circuit that uses two random field elements $(z_{\mathsf{fi}}, z_{\mathsf{tp}})$ (defined in Section 3.2) to combine (aka "pack") each triple of node values into one field element by taking a random linear combination, as follows:

$$\mathsf{path}^{(p)}_{\bar{x}_{i,j}} = [(\overline{\mathsf{id}}_0 + \overline{\mathsf{fi}}_0 \cdot z_{\mathsf{fi}} + \overline{\mathsf{thr}}_0 \cdot z_{\mathsf{tp}}), \dots, (\overline{\mathsf{id}}_{h-2} + \overline{\mathsf{fi}}_{h-2} \cdot z_{\mathsf{fi}} + \overline{\mathsf{thr}}_{h-2} \cdot z_{\mathsf{tp}}), (\overline{\mathsf{id}}_{h-1} + \overline{\mathsf{pred}}_{T,x} \cdot z_{\mathsf{tp}})]$$

Then, our circuit uses a random field element r_{mst} (defined in Section 3.2) to compute the evaluation at r_{mst} of a low-degree polynomial with its roots being the field elements from all the "packed" paths $\mathsf{path}^{(p)}_{\bar{x}_{i,j}}$ for every $\bar{x}_{i,j}$. That is,

$$\prod_{i=1}^{n} \prod_{k=0}^{h-1} \left(r_{\mathsf{mst}} - \mathsf{path}^{(p)}_{\bar{x}_{i,j}}[k] \right)$$

Then, our circuit computes the "packed" tree $T_j^{(p)}$ defined as follows:

$$\begin{aligned} T_j^{(p)} = \Big[& (0 + \mathsf{fi}_0 \cdot z_{\mathsf{fi}} + \mathsf{thr}_0 \cdot z_{\mathsf{tp}}), \dots, \left((2^{(h-1)} - 2) + \mathsf{fi}_{2^{(h-1)}-2} \cdot z_{\mathsf{fi}} + \mathsf{thr}_{2^h-2} \cdot z_{\mathsf{tp}} \right), \\ & \left((2^{(h-1)} - 1) + \mathsf{pred}_{2^{(h-1)}-1} \cdot z_{\mathsf{tp}} \right), \dots, \left((2^h - 2) + \mathsf{pred}_{2^h-2} \cdot z_{\mathsf{tp}} \right) \Big] \end{aligned}$$

Then, our circuit using the binary decomposition of ct_k evaluates a low-degree polynomial with its roots being the field elements from all the "packed" tree nodes. Below we give the final result of this circuit computation by skipping the binary decomposition step,

$$\prod_{k=0}^{2^{h+1}-2} \left(r_{\mathsf{mst}} - T_j^{(p)}[k] \right)^{\mathsf{ct}_k}$$

To compute the above quantity, our circuit uses the binary decomposition of ct_k defined as $\mathsf{BitDecomp}(\mathsf{ct}_k) = (\mathsf{ct}_{(k,0)}, \dots, \mathsf{ct}_{(k,\ell)})$, as follows:

$$\prod_{k=0}^{2^{h+1}-2} \left(r_{\mathsf{mst}} - T_j^{(p)}[k] \right)^{\mathsf{ct}_k} = \prod_{k=0}^{2^{h+1}-2} \prod_{s=0}^{\ell-1} \left(\mathsf{ct}_{(k,s)} \cdot \left(r_{\mathsf{mst}} - T_j^{(p)}[k] \right)^{2^s} + \left(1 - \mathsf{ct}_{(k,s)}\right) \right)$$

Finally, our circuit outputs

$$\prod_{i=1}^{n} \prod_{k=0}^{h-1} \left(r_{\mathsf{mst}} - \mathsf{path}^{(p)}_{\bar{x}_{i,j}}[k] \right) - \prod_{k=0}^{2^{h+1}-2} \left(r_{\mathsf{mst}} - T_j^{(p)}[k] \right)^{\mathsf{ct}_k}$$

which is expected to be zero, if every $\mathsf{path}_{\bar{x}_{i,j}}$ contains nodes only from T. As in Section 3.3.1, a similar usage of the Schwartz-Zippel lemma applies here for the above difference.

3.4 Claim aggregation

Here, we present a heuristic method to aggregate claims in the context of GKR for our specific application to decision tree inference.

A *claim* is an assertion to be proven about the evaluation of a multilinear polynomial $\widetilde{V}_i : \mathbb{F}^n \to \mathbb{F}$ associated with Layer i of the circuit at a point $u \in \mathbb{F}^n$. During the GKR protocol the proof of a claim $\widetilde{V}_i(u) = c$ is reduced to, perhaps multiple, claims on multilinear polynomials $\widetilde{V}_j$ for previous layers $j > i$. The goal of *claim aggregation* is that of reducing a sequence of m claims $\widetilde{V}_i(u_0) = c_0, \widetilde{V}_i(u_1) = c_1, \ldots, \widetilde{V}_i(u_{m-1}) = c_{m-1}$ to a single claim on the same layer $\widetilde{V}_i(u^*) = c^*$ such that the soundness of the resulting claim implies the soundness of all m original claims. That way, we avoid an exponential blowup in the number of claims, and instead we end up with a single claim per layer to be proven using the sumcheck protocol. For the remainder of this section we focus on a single layer and will thus omit the subscript i on $\widetilde{V}$.

One method for solving this problem is presented in [Tha22, Section 4.5.2] and can be summarized as follows: both the prover and the verifier use Lagrange interpolation to compute the unique, univariate polynomial $P : \mathbb{F} \to \mathbb{F}^n$ of degree at most $m - 1$ such that $P(0) = u_0, P(1) = u_1, \ldots, P(m-1) = u_{m-1}$. The prover computes the polynomial $Q(x) \coloneqq \widetilde{V}(P(x))$ and sends it over to the verifier. Recall that $\widetilde{V}$ is multilinear on n variables, hence $Q(x)$ is of degree at most $(m-1) \cdot n$. The verifier first verifies that $Q(0) = c_0, \ldots, Q(m-1) = c_{m-1}$, then picks a uniformly random challenge $r^* \in \mathbb{F}$ and sets $u^* = P(r^*), c^* = Q(r^*)$. The Schwartz-Zippel lemma implies that if the prover successfully convinces the verifier for the validity of the claim $\widetilde{V}(u^*) = c^*$, then with high probability the verifier is convinced that Q and $\widetilde{V} \circ P$ represent the same polynomials and hence convinced for the validity of all of the original claims.

The computational complexity of the above procedure depends on how we choose to represent polynomials. In our implementation, we are using almost exclusively evaluations-based representations for polynomials and multilinear extensions. For example, to represent a univariate polynomial $P : \mathbb{F} \to \mathbb{F}$ of degree d, we store a vector of $d+1$ field elements $[P(0), P(1), \ldots, P(d)]$ which uniquely define P. To store the multilinear extension $\widetilde{V} : \mathbb{F}^n \to \mathbb{F}$ of a boolean function $V : \{0,1\}^n \to \mathbb{F}$, we store all 2^n evaluations of V over the boolean hypercube. Under this representation, we implement claim aggregation in time $O(n \cdot m \cdot 2^n)$ as follows: to represent $\widetilde{V} \circ P$, we need to evaluate it at $O(d)$ distinct points where d is its degree, and we've argued previous that $d = O(n \cdot m)$. Each polynomial evaluation can be done in time linear to the representation of the polynomials involved, in particular in time $O(2^n)$ due to $\widetilde{V}$.

Our first heuristic is to notice that if all points $u_0, \ldots, u_{m-1}$ agree on some coordinate j, then the polynomial $P_j(x)$ of the j-th coordinate of $P(x)$ is constant. This observation provides a better upper bound on the degree of $\widetilde{V} \circ P$ resulting in fewer evaluations for its representation[2]. In general, if the input to the claim aggregation procedure consists of m claims such that there exists a set S of $k \leq n$ coordinates on which all claims agree on, i.e. $\forall j \in S : u_{0j} = u_{1j} = \ldots = u_{(m-1)j}$, then we can perform claim aggregation in time $O((n-k) \cdot m \cdot 2^n)$ by noticing that the total degree of $\widetilde{V} \circ P$ is now bounded by $O((n-k) \cdot m)$. Further, since $P_j(x)$ is constant for all $j \in S$, we can guarantee that the resulting claim $\widetilde{V}(u^*) = c^*$ has the property that $\forall j \in S : u^*_j = u_{0j} = u_{1j} = \ldots = u_{(m-1)j}$, i.e. the constant coordinates are preserved after the aggregation. Let's call this procedure "elementary claim aggregation".

For general GKR circuits whose layer MLEs are represented using addition/multiplication gates as in Equation 2.3, it is unlikely that all claims on a given layer would agree on some coordinate;

[2] Besides improving performance, a better upper bound on the degree of $\widetilde{V} \circ P$ improves the protocol security since it makes the work of a cheating prover even harder by restricting them to lower degree polynomials.

the summation includes terms of the form $\widetilde{V}_{i+1}(x), \widetilde{V}_{i+1}(y)$ and the sumcheck protocol binds x, y to different random vectors. However, in structured circuits as the ones discussed in Section 2.1.2, the layer MLEs can be described as in Equation 2.6 in which the arguments to the MLEs of the previous layer share a common suffix of iterated variables on the summation. As a result, the sumcheck protocol generates claims whose point vectors have a special structure: they agree on a suffix of random challenges and further, they include a prefix of fixed bits.

Consider the following example of 8 claims on some layer k, where $r_1, \ldots, r_{n-2}, r'_1, \ldots, r'_{n-2}, c_0, \ldots, c_7 \in \mathbb{F}$ are arbitrary field elements:

$$\begin{aligned}
\widetilde{V}(0, 0, r_1, \ldots, r_{n-2}) &= c_0 \\
\widetilde{V}(0, 1, r_1, \ldots, r_{n-2}) &= c_1 \\
\widetilde{V}(1, 0, r_1, \ldots, r_{n-2}) &= c_2 \\
\widetilde{V}(1, 1, r_1, \ldots, r_{n-2}) &= c_3
\end{aligned}$$

$$\begin{aligned}
\widetilde{V}(0, 0, r'_1, \ldots, r'_{n-2}) &= c_4 \\
\widetilde{V}(0, 1, r'_1, \ldots, r'_{n-2}) &= c_5 \\
\widetilde{V}(1, 0, r'_1, \ldots, r'_{n-2}) &= c_6 \\
\widetilde{V}(1, 1, r'_1, \ldots, r'_{n-2}) &= c_7
\end{aligned}$$

Collections of claims of this form are commonly found in structured circuits for which a Layer i is allowed to have connections to any Layer j for $j > i$, not just the immediatelly preceeding ones. In the example above, Layer k received claims from two separate layers $i, j < k$ as follows: the sumcheck protocol on layer i produced the upper set of four claim on layer k, while the sumcheck protocol on layer j produced the lower set of four claims. Notice how claims produced from the same layer agree on the random challenges as a result of the regular structure of the circuit as reflected in Equation 2.6.

Aggregating all $m = 8$ claims in one round using the the elementary claim aggregation algorithm described above requires time $n \cdot m \cdot 2^n = 8 \cdot n \cdot 2^n$, since all the columns disagree in at least one coordinate.

But notice how the first four claims all agree on a suffix of length $k = n-2$ of their coordinates and hence the elementary claim aggregation procedure has an advantage when applied to this subset of claims. Following the previous discussion, we see that aggregating just the first four claims requires time proportional to $(n-k) \cdot m \cdot 2^n = 2 \cdot 4 \cdot 2^n = 8 \cdot 2^n$ and results in an aggregate claim of the form $\widetilde{V}(r^*_a, r^*_b, r_1, \ldots, r_{n-2}) = c^*_0$. Similarly, aggregating the last four claims takes the same amount of time and results in a claim of the form $\widetilde{V}(r^*_c, r^*_d, r'_1, \ldots, r'_{n-2}) = c^*_1$. Aggregating the resulting $m = 2$ claims requires time proportional to $n \cdot m \cdot 2^n = 2 \cdot n \cdot 2^n$. In total, using this two-stage procedure we aggregated all original claims into one in time proportional to $2 \cdot 8 \cdot 2^n + 2 \cdot n \cdot 2^n = (16 + 2n) \cdot 2^n$. For large enough n, the latter method happens to be more efficient in this instance.

This example suggests that, in general, the optimal aggregation strategy might consist of many stages of elementary claim aggregation steps on subsets of claims. We can represent a claim aggregation strategy as a rooted tree whose leaves are the original claims and internal nodes represent intermediate claims that result from running the elementary claim aggregation procedure on its children. We can then associate a cost to each non-leaf node capturing the complexity of running elementary claim aggregation on its children which is given by the formula $(m-k) \cdot m \cdot 2^n$ that we derived earlier. The total cost of aggregation is the sum of the costs of all non-leaf nodes of the tree.

We have evidence to believe that the problem of optimal claim aggregation as formulated in the previous paragraph is NP-hard. However, we've found that in practice, a heuristic that works particularly well for structured circuits is the following: aggregate using a two-stage strategy by first grouping claims based on the layer from which they originated — recall that GKR propagates claims from layers closer to the output to layers closer to the input. Then aggregate the claims within each group using the elementary claim aggregation procedure (stage 1) and finally aggregate all the claims that resulted from stage 1 to one final claim (stage 2). The intuition behind this is that claims originating from the same layer have a higher likelihood of agreeing on some coordinate as a result of the regular structure of some parts of our circuits, allowing us to take advantage of the speed up achievable by the heuristic in the elementary claim aggregation procedure.

3.5 Adapting from the Libra and Giraffe works

In our data-parallel decision tree circuit, the path consistency checking circuit (3.3.2) requires the usage of wiring predicates, and in particular both addition and multiplication data parallel gate predicates. In other words, we require equations of the form:

$$V_i(p_1, p_2) = f_1(p_1, x, y)\big(V_{i+1}(p_2, x) \diamond V_{i+1}(p_2, y)\big)$$

- $f_1 : \{0,1\}^{s_i + 2s_{i+1}} \mapsto \{0,1\}$ represents the "gate function", which takes in tuples of the form (p_1, x, y) where p_1 is a binary representation of the output gate label, and x and y are binary representations of the two input gate labels.
- "$\diamond$" represents the "gate operation" (e.g. addition or multiplication) that we apply to the input gate values in order to compute the output gate value.
- p_2 is the binary representation of the circuit copy label.
- $V_i(p_2, x) : \{0,1\}^{b+s_i} \mapsto \mathbb{F}$ is the value of the x^{th} gate in the ${p_2}^{\text{th}}$ copy of the circuit, and similarly for $V_i(p_2, y)$.

Then, the multilinear extension of V_i over $(g_2, g_1) \in \mathbb{F}^{b+s_i}$ is:

$$\widetilde{V}_i(g_2, g_1) = \sum_{(p_2, p_1, x, y) \in \{0,1\}^{b+s_i+2s_{i+1}}} \widetilde{\beta}(g_2, p_2) \cdot \widetilde{\beta}(g_1, p_1) \cdot f_1(p_1, x, y) \cdot \big(\widetilde{V}_{i+1}(p_2, x) \diamond \widetilde{V}_{i+1}(p_2, y)\big)$$

where $\widetilde{\beta}$ is the Lagrange basis polynomial as defined in Section 2.1.

We extend the linear-time prover techniques from Libra [XZZ$^+$19] in order to compute sumcheck messages for the above summation in $O(2^{b+s_i})$ for data-parallel gate MLEs. Additionally, we take inspiration from Giraffe [WJB$^+$17] to determine the order in which we bind the bits of the MLE. We first describe our algorithm for binding the data-parallel bits for each sumcheck round (3). Then, we analyze the runtime of each of these rounds and provide motivation for binding the bits in this particular order.

The algorithms in Libra [XZZ$^+$19] specifically focus on the non-data-parallel gate predicates. Semantically, the sumcheck rounds are split into two "phases": one phase binding the x variables, the other binding the y variables. Each step involves a pre-processing step which computes the summation over all the variables that are not currently being bound. Therefore, the MLEs in which sumcheck messages are computed over during each phase are over s_i variables rather than $2s_i$ variables. Note that beceause the p_2 variables are shared with MLEs containing x variables and y variables, we cannot apply the same pre-processing techniques as in the non-data-parallel case. We specifically look at the case where the gate operation, $\diamond$, is a multiplication. We rewrite the summation as follows:

$$\widetilde{V}_i(g_2, g_1) = \sum_{p_2 \in \{0,1\}^b} \widetilde{\beta}(g_2, p_2) \cdot \left(\sum_{(p_1, x, y) \in \{0,1\}^{s_i + 2s_{i+1}}} \widetilde{\beta}(g_1, p_1) \cdot f_1(p_1, x, y) \cdot \widetilde{V}_{i+1}(p_2, x) \cdot \widetilde{V}_{i+1}(p_2, y) \right)$$

Furthermore, let $\mathcal{N}_{(p_1,x,y)}$ be the set of tuples such that $f_1(p_1, x, y) \neq 0$. Then we can implicitly factor in the $f_1(p_1, x, y)$ term of the above expression:

$$\widetilde{V}_i(g_2, g_1) = \sum_{p_2 \in \{0,1\}^b} \widetilde{\beta}(g_2, p_2) \cdot \left(\sum_{(p_1, x, y) \in \mathcal{N}_{(p_1,x,y)}} \widetilde{\beta}(g_1, p_1) \cdot \widetilde{V}_{i+1}(p_2, x) \cdot \widetilde{V}_{i+1}(p_2, y) \right)$$

When binding the p_2 variables, the evaluations sent from the prover to the verifier would be of the following univariate polynomial (as defined in section 2.1), assuming we are on the jth round of sumcheck, $j < b$, and for the rounds $i < j$, p_{2_i} is bound to r_i:

$$\begin{aligned} h(X) = & \\ \sum_{(p_{2_{[j+1:b]}}) \in \{0,1\}^{b-j}} & \widetilde{\beta}(g_2, (r_1, \ldots, r_{j-1}, X, p_{2_{j+1}}, \ldots, p_{2_b})) \quad \cdot \\ \Bigg(\sum_{(p_1, x, y) \in \mathcal{N}_{(p_1,x,y)}} & \widetilde{V}_{i+1}((r_1, \ldots, r_{j-1}, X, p_{2_{j+1}}, \ldots, p_{2_b}), x) \quad \cdot \\ & \widetilde{V}_{i+1}((r_1, \ldots, r_{j-1}, X, p_{2_{j+1}}, \ldots, p_{2_b}), y) \cdot \widetilde{\beta}(g_1, p_1) \Bigg) \end{aligned}$$

Given this form-factor, we propose the following algorithms in pseudocode in order to bind the first b bits (the p_2 bits) of the expression:

Algorithm 2: ComputeSuccessors

```
Input: mle, index, num_evaluations
Result: linear interpolation of mle at index over num_evaluations number of evaluations
// select the index-th coefficient of the MLE, assuming coefficients are ordered in
   little-endian
first = mle[index];
second = mle[index];
linear_difference = second - first;
linear_interpolation_array = [first, second];
current_evaluation = second;
for eval_num = $2, \ldots,$ num_evaluations do
    next_linear_evaluation = current_evaluation + linear_difference;
    linear_interpolation_array.append(next_linear_evaluation);
    current_evaluation = next_linear_evaluation;
end
return linear_interpolation_array;
```

Algorithm 3: Libra-Giraffe – Round of Sumcheck for Binding Data-parallel Bits

```
// assume we are on the $j$th round of sumcheck, $j < b$
// f1 $= \mathcal{N}_{(p_1,x,y)}$
// f2 $= V_{i+1}(r_1, \ldots, r_j, p2_{[j+1:b]}, x)$
// f3 $= V_{i+1}(r_1, \ldots, r_j, p2_{[j+1:b]}, y)$
// $(g1, g2)$ are equivalent to $g_1, g_2$ in the equations above
// num_dataparallel_bits $= b - j$
Input: $f1, f2, f3, g2, g1$, num_data_parallel_bits
// evaluations of $h(X)$ as defined above
Result: [h(0), h(1), h(2), h(3)]
// compute Lagrange basis MLEs for $g_2$ and $g_1$
beta_g2 $= \widetilde{\beta}(g_2, p_2)$;
beta_g1 $= \widetilde{\beta}(g_1, p_2)$;
// in the case of $\circ$ being a multiplication, we have a degree 3 univariate, so we need 4
   evaluations to fit onto a unique curve
num_evaluations = 4;
for p2_idx $= 0, \ldots, 2^{\text{num\_data\_parallel\_bits}}$ do
    outer_sum_evaluations $= [0, 0, 0, 0]$;
    beta_g2_successors = ComputeSuccessors(beta_g2, p2_idx $\cdot$ 2, num_evaluations);
    for $(p_1, x, y) \in f1$ do
        inner_sum_evaluations $= [0, 0, 0, 0]$;
        // we assume coefficients of MLEs are stored in little-endian, so we do the
           appropriate index shifting
        f2_successors = ComputeSuccessors(f2, p2_idx $\cdot 2 + (x \cdot 2^{\text{num\_data\_parallel\_bits}})$,
          num_evaluations);
        f3_successors = ComputeSuccessors(f3, p2_idx $\cdot 2 + (y \cdot 2^{\text{num\_data\_parallel\_bits}})$,
          num_evaluations);
        // select the $p_1$ coefficient of the beta_g1 MLE, assuming coefficients are ordered in
           little-endian.  this is a constant function over $p_2$ so we repeat 4 times
        beta_g1_successors = [beta_g1[$p_1$]; 4];
        inner_sum_successors = elementwise product of f2_successors, f3_successors, and
          beta_g1_successors;
        // += refers to elementwise addition
        inner_sum_evaluations += inner_sum_successors;
    end
    outer_sum_successors = elementwise product of inner_sum_evaluations,
      beta_g2_successors;
    // += refers to elementwise addition
    outer_sum_evaluations += outer_sum_successors;
end
return outer_sum_evaluations;
```

We now analyze the runtime of the above algorithm, and then compare this to the runtime of possible alternative algorithms in order to motivate our decision to bind the data-parallel bits first. Note that for the first b rounds of sumcheck, using our algorithm (3), we iterate through all indices of the MLEs corresponding to the data-parallel bits, which is 2^{b-j} iterations for the jth round, $j < b$. Within each iteration, we also compute the summation through all of the "nonzero gates" ($\mathcal{N}_{(p_1,x,y)}$) in the inner loop, which is at most size $2^{s_{i+1}}$ assuming the sparsity of the gate function f_1. Therefore the runtime of each round is $O(2^{b-j+s_{i+1}})$. Over all b data-parallel rounds, this evaluates to:

$$\sum_{j=0}^{b} O(2^{b-j+s_{i+1}}) = O(2^{b+s_{i+1}})$$

Once the p_2 bits are fully bound to our challenge r_{p_2} (i.e. after the first b rounds of sumcheck), our multilinear extension is now:

$$\widetilde{V}_i(g_2, g_1) = \sum_{(p_1,x,y)\in\{0,1\}^{s_i+2s_{i+1}}} \widetilde{\beta}(g_2, r_{p_2}) \cdot \widetilde{\beta}(g_1, p_1) \cdot f_1(p_1, x, y) \cdot \widetilde{V}_{i+1}(r_{p_2}, x) \circ \widetilde{V}_{i+1}(r_{p_2}, y)$$

Note that $\widetilde{\beta}(g_2, r_{p_2})$ is a constant, and our summation is now over $s_i + 2s_{i+1}$ variables. As noted in Giraffe [WJB+17], this is a reduction to the non-data-parallel GKR equation and we can now use the linear-time prover tricks in Libra [XZZ+19] to achieve a $2\sum_{k=0}^{s_{i+1}} 2^{s_{i+1}-k} = O(2^{s_{i+1}})$ runtime for the rest of the $2s_{i+1}$ rounds binding x and y.

Alternatively, if we were to bind the x and y variables first, we cannot apply the same preprocessing techniques as done before binding the x and y variables in Libra. This is for the exact same reason in which we cannot apply the pre-processing techniques before the first b rounds of sumcheck in our algorithm – because the p_2 bits are shared among both MLEs contributing to the output gate value, it is impossible to compute the sum of the variables not being bound beforehand in order to reduce the number of variables of the MLEs in the expression. Therefore, the runtime for the first $2s_{i+1}$ rounds of sumcheck (without loss of generality, assuming we bind the x variables, and then the y variables) would be:

$$2\sum_{k=0}^{s_{i+1}} 2^{b+s_{i+1}-k} = O(2^{s_{i+1}+b}).$$

For the last b rounds, it would be $\sum_{j=0}^{b} 2^{b-j} = O(2^b)$. While asymptotically, the runtimes of both algorithms are the same, we do a more detailed cost analysis on the exact runtimes to show that binding the p_2 variables first is indeed the most efficient strategy. Note that the exact runtime of our presented algorithm is:

$$\leq \sum_{j=0}^{b} 2^{b-j+s_{i+1}} + 2\sum_{k=0}^{s_{i+1}} 2^{s_{i+1}-k}$$

$$= \sum_{t=0}^{b+s_{i+1}} 2^{b+s_{i+1}-t} + \sum_{k=0}^{s_{i+1}} 2^{s_{i+1}-k}.$$

and the exact runtime of the alternative binding strategy is:

$$2\sum_{k=0}^{s_{i+1}} 2^{b+s_{i+1}-k} + \sum_{j=0}^{b} 2^b - j$$

$$= \sum_{t=0}^{b+s_{i+1}} 2^{b+s_{i+1}-t} + \sum_{k=0}^{s_{i+1}} 2^{s_{i+1}+b-k}.$$

Therefore binding the x and y variables first is slower by an additive factor of $O(2^{b+s_{i+1}})$. We omit the analysis of binding the p_2 variables in the middle of binding the x and y variables as it results in a similar additive slowdown compared to our proposed algorithm. In Section 4.1 we provide benchmarks of this data-parallel extension of Libra.

4 Benchmarks

We present a variety of benchmarks showing prover time scaling against other factors for Libra-Giraffe and our overall Verifiable Decision Forest circuit. Note that all benchmarks were run over a Macbook Pro M2, with 32GB RAM, 12 cores, and full parallelism. Recall that Libra-Giraffe details a sumcheck prover over a dataparallel circuit, with each subcircuit wired in an arbitrary way via binary addition and multiplication gates. Under the Libra-Giraffe algorithm, we expect the prover's runtime across all rounds of sumcheck for a single layer of the dataparallel circuit to be linear with respect to the number of dataparallel circuit copies, as the prover must roughly perform at least the work across a single circuit copy, but duplicated across all parallel copies. Additionally, we expect the prover's runtime to be linear with respect to the size of the previous layer, as each of these values contribute to the expression on the RHS of 3.5 in the form of the $\widetilde{V}_{i+1}(p_2, x)$ and $\widetilde{V}_{i+1}(p_2, y)$ terms, and finally, we expect the prover's runtime to be linear with respect to the number of gates between the two layers (assumed here to be the number of values within the current layer), as each of these gates contributes to the values within the current layer and should be accounted for. We give a more formal statement below:

4.1 Libra-Giraffe Benchmarks

In accordance with Section 3.5, we show that the proof generation time for a sumcheck layer over dataparallel $\mathsf{add}_i(z, x, y)$ and $\mathsf{mult}_i(z, x, y)$ gates (and note that both of these gate terms are the $f_1(p_1, x, y)$ term described in Section 3.5) is *linear* with respect to the following parameters:

- The layer size. That is, 2 to the power of the number of x and y bits. Note that in general, the numbers of x and y bits are always equal.

- The number of dataparallel copies of the circuit. That is, 2 to the power of the number of p_2 dataparallel bits.

- The number of nonzero gates. That is, the size of $\mathcal{N}_{(p_1,x,y)}$.

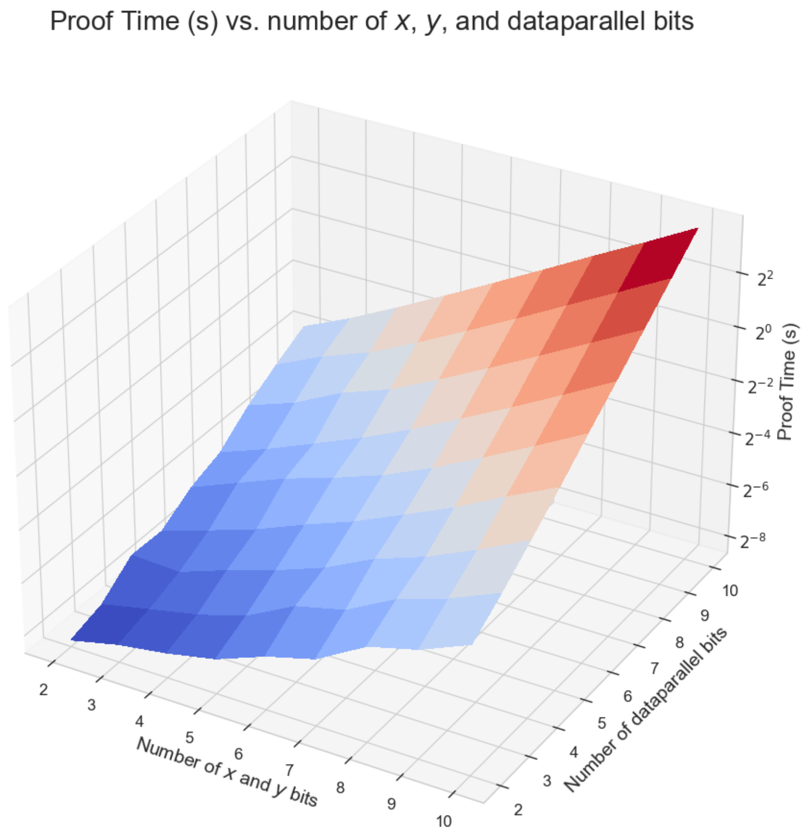


Figure 1: *(Log-scale) Libra-Giraffe prover time scaling as a function of number of the layer size, i.e. the number of (x, y) bits, and the number of dataparallel p_2 bits.*

We see that in accordance with our prover runtime analysis of $O(2^{s_i+b})$, the Libra-Giraffe prover achieves a linear relationship between $2^{|x|} = 2^{|y|}$ and the total prover time in the above plot (recall that $|x| = |y| = s_i$). For a fixed number $|b|$ of dataparallel bits, the prover runtime of 2^{s_i+b} is a constant multiple (2^b) of $2^{s_i} = 2^{|x|} = 2^{|y|}$ – exactly the trend shown in the above plot.

Similarly, for a fixed layer size s_i we have that 2^b, the total number of subcircuit copies, is a constant multiple, 2^{s_i}, from the prover runtime, $O(2^{s_i+b})$, and thus the linear relationship as seen above is what was expected.

4.2 Claim Aggregation Benchmarks

We see below benchmarks for generating GKR proofs over circuits of various sizes, with a variety of optimizations turned on/off.

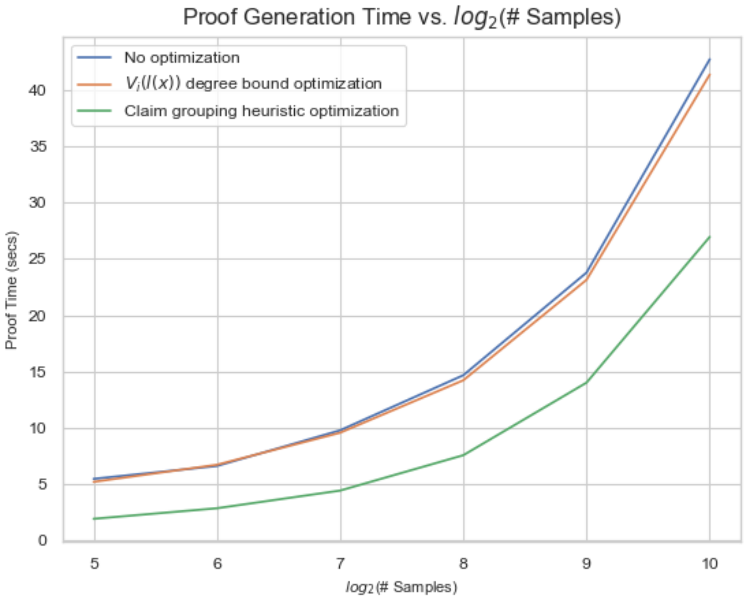
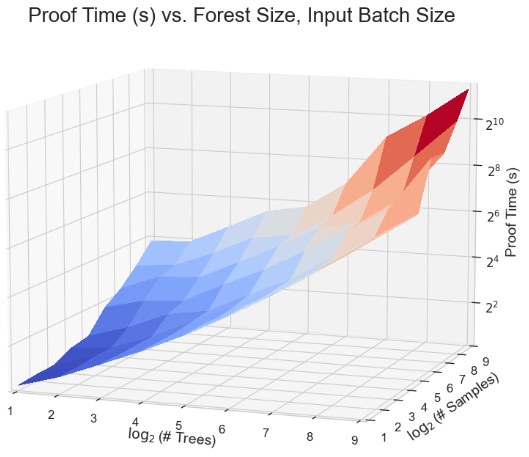


Figure 2: *Proof time vs. circuit size (as determined by number of samples) and type of optimization used, if any. Note that the number of trees was kept constant, i.e. inference over 2 trees was proven for each batch of samples.*

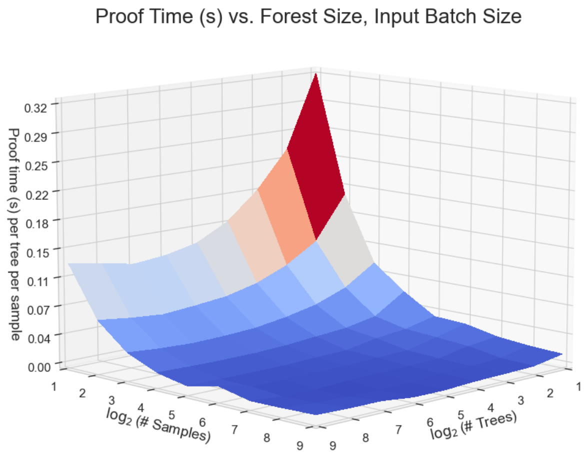
Note that in the above figure, the "$V_i(l(x))$ degree bound optimization" plot refers to the "first heuristic" described in section 3.4, where all claims on a certain layer, regardless of origin, are checked for columns in which *all* challenge values match, and a constant polynomial is interpolated in that case. The "claim grouping heuristic optimization" refers to the one described in the last paragraph within that same section, i.e. the two-layer claim aggregation strategy where claims are first grouped by the layer within which they originated. We see that while the first heuristic hardly makes a difference in the overall proof generation time due to the heterogeneity of the claims on most layers, resulting in few constant coordinate columns, the heuristic cuts the prover runtime by anywhere from 40-60%, and represents a substantial saving in a complex, practical circuit as the Verifiable Decision Forest circuit.

4.3 Verifiable Decision Forest Circuit Benchmarks

We display a variety of plots detailing the relationship between (tree height, sample length, num samples, num trees) and prover runtime. As a result of our circuit optimizations (e.g. multiset circuits for inputs and tree nodes), we expect the prover runtime to scale even sublinearly in certain cases as described below.

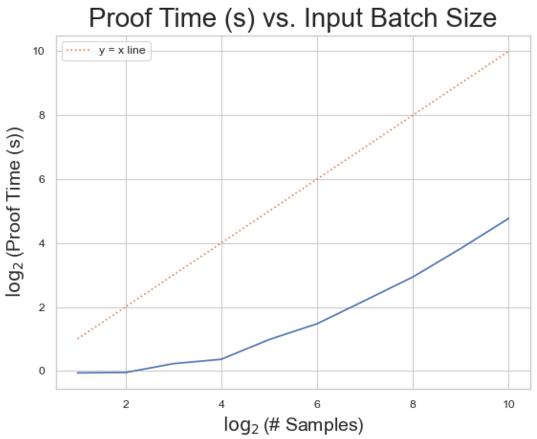


(a) *Proof time against input batch size and number of trees in forest*

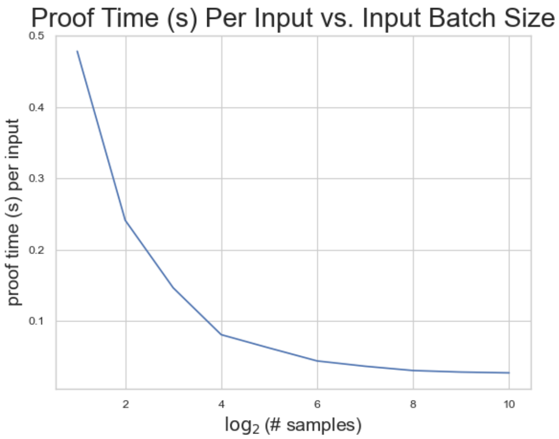


(b) *Proof time per number of samples per number of trees against input bath size and forest size*

First, we plot the relationships between (number of trees, number of input samples) against the prover runtime, and also plot the runtime-per-tree-per-sample (prover runtime divided by the product of the number of trees and number of samples), to measure how our prover scales as the decision forest size grows and the input batch size grows.

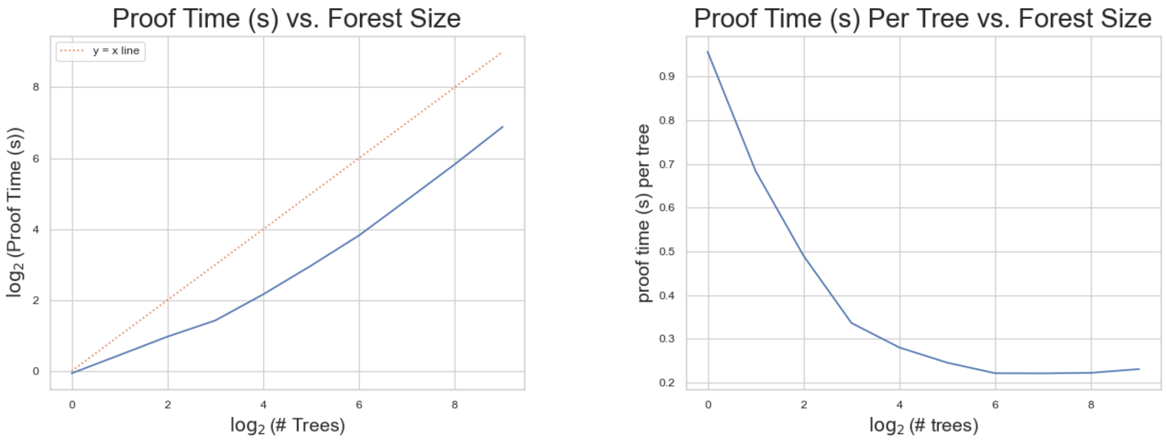


(a) *Proof time against input batch size*



(b) *Proof time per batch size against input batch size*

As we can see on the LHS graph, the prover time grows far sublinearly with respect to the number of samples, a trend which is amplified by the RHS graph. This shows in particular the nice scaling properties of the multiset circuit, which does not increase in size as the number of inputs increases (indeed, only the multiplicity *values* increase).



(a) *Proof time against number of trees in forest* (b) *Proof time per forest size against number of trees*

Similarly, we see that the prover runtime scales nicely with respect to the number of trees within the decision forest. This is a result of the input multiset circuit, whose size is also independent of the number of trees and only relies on the number of inputs within the batch.

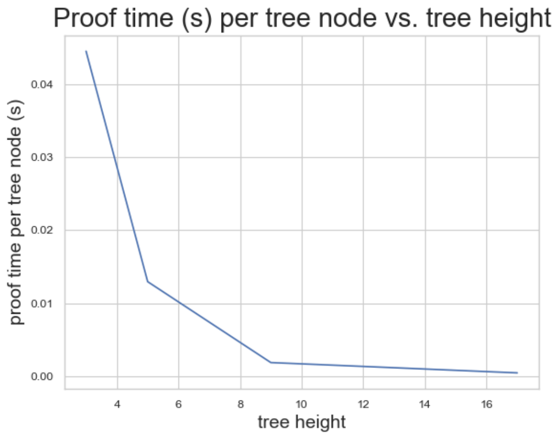
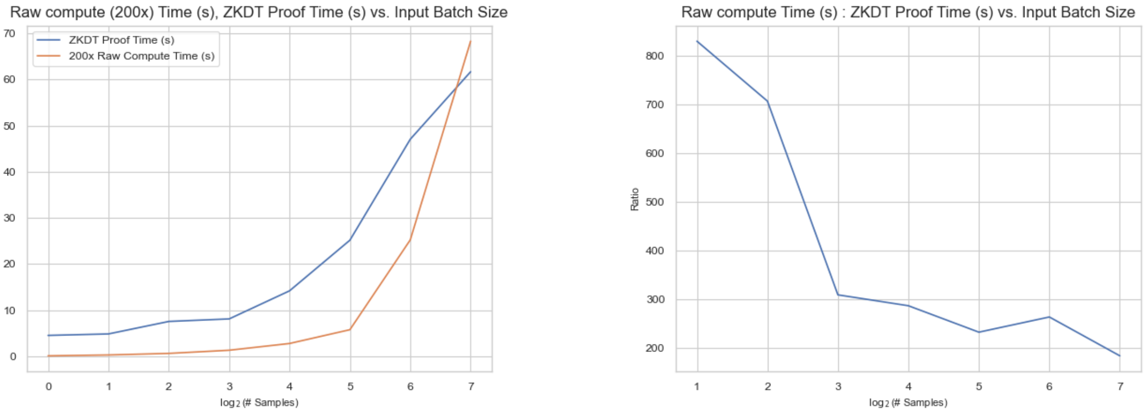


Figure 6: *Proof time per number of tree nodes against tree height*

Finally, the above graph describes the prover runtime for trees of varying *size*, averaged over the total number of nodes in the tree. We see here that because of the path consistency check, which scales only with the *height* of the tree (which is logarithmic in the total tree size), the prover also scales well with individually larger trees.

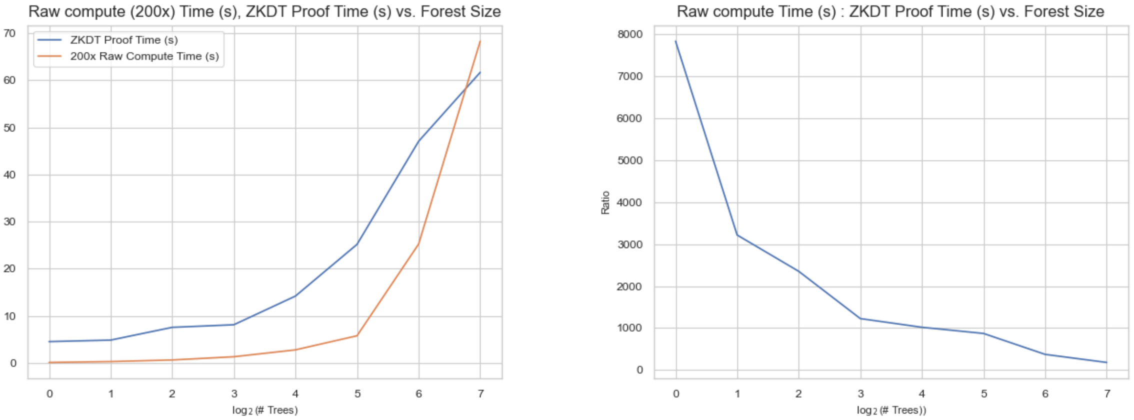
4.4 Raw Computation vs. Proving Benchmarks

We perform a final experiment to highlight the efficiency of our Verifiable Decision Forest circuit, by comparing the total time spent proving (GKR and Ligero) to performing the raw decision forest inference itself in a multithreaded Rust environment.



(a) *Remainder proof time vs. increasing input batch size compared to 200x raw-computation time of inference on an input batch and forest of the same size*

(b) *Ratio of raw computation of a decision forest to Remainder proof time vs. increasing input batch size*



(a) *Remainder proof time vs. increasing input batch size compared to 200x raw-computation time of inference on a forest and input batch of the same size*

(b) *Ratio of raw computation of a decision forest to Remainder proof time vs. increasing decision forest size*

We note in both cases that the ratio drops quite quickly, bottoming out at around 180x proof generation overhead (i.e. the cost of computing a GKR and Ligero proof is just 180x slower than performing the inference computation itself), allowing such a system to be used even for production-scale decision forest models (indeed, a version of the prover and Verifiable Decision Forest circuit described in this paper has already been deployed for a decision forest of over 2500 trees).

5 Future Directions

Although the Remainder GKR prover is highly optimized for speed and our Verifiable Decision Forest circuit is written in a scalable fashion, many research directions remain:

- We explored the runtime efficiency of e.g. Libra-Giraffe and implemented a linear-time structured [Tha13] circuit prover, but we did not foucs on the memory footprint of such a prover. What would the memory footprint of proving the Verifiable Decision Forest circuit be over a memory-optimized prover, e.g. one in which GKR layers only have their circuit values materialized during proving, rather than having all of their circuit values present in RAM at once?

- We run our benchmarks with an implementation of the Ligero polynomial commitment scheme (PCS) for the GKR circuit's input layer, instantiated using the Poseidon hash function. However, this results in relatively large proofs for the input layer (specifically, Ligero proofs require sending at least 200 "columns" of field elements to the verifier, and yield concrete proofs of over 1MB). We leave as future work the use of a different PCS, such as for example the one in [ZXZS20] or the one in [KT23], that would yield similar prover times while reducing the concrete proof size.

- We investigated several claim aggregation strategies, but only tested these empirically against our Verifiable Decision Forest circuit. Indeed, different claim aggregation strategies may work better for differently-structured circuits.

- GKR remains a natural option for other types of machine learning models, in particular large, layered neural networks – indeed, the [Tha13] matrix multiplication sumcheck protocol proves precisely the dense layer of feedforward neural networks – could we see even lower prover overheads for such models?

References

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Heidelberg, August 2013.

[BCR+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Heidelberg, May 2019.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[CFF+23] Matteo Campanelli, Antonio Faonio, Dario Fiore, Tianyu Li, and Helger Lipmaa. Lookup arguments: Improvements, extensions and applications to zero-knowledge decision trees. Cryptology ePrint Archive, Paper 2023/1518, 2023. https://eprint.iacr.org/2023/1518.

[DBK$^+$20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[DP23] Benjamin E. Diamond and Jim Posen. Proximity testing with logarithmic randomness. Cryptology ePrint Archive, Paper 2023/630, 2023. https://eprint.iacr.org/2023/630.

[FQZ$^+$21] 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, Report 2021/087, 2021. https://eprint.iacr.org/2021/087.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

[git] Github copilot. https://github.com/features/copilot. Accessed: 2024-02-06.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 113–122. ACM Press, May 2008.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4), sep 2015.

[GKR$^+$19] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Paper 2019/458, 2019. https://eprint.iacr.org/2019/458.

[GKR$^+$21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 519–535. USENIX Association, August 2021.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

[KHSS22] Daniel Kang, Tatsunori B. Hashimoto, Ion Stoica, and Yi Sun. Scaling up trustless dnn inference with zero-knowledge proofs. *ArXiv*, abs/2210.08674, 2022.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, page 723–732, New York, NY, USA, 1992. Association for Computing Machinery.

[KRGH23] Harsh Kumar, David M. Rothschild, Daniel G. Goldstein, and Jake Hofman. Math education with large language models: Peril or promise? http://dx.doi.org/10.2139/ssrn.4641653, Dec 2023.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[KT23] Tohru Kohrita and Patrick Towa. Zeromorph: Zero-knowledge multilinear-evaluation proofs from homomorphic univariate commitments. Cryptology ePrint Archive, Paper 2023/917, 2023. https://eprint.iacr.org/2023/917.

[Lab23] Modulus Labs. The cost of intelligence: Proving machine learning inference with zero-knowledge. https://drive.google.com/file/d/1tylpowpaqc0hKQtYolPlqvx6R2Gv4IzE/view?usp=sharing, Jan 2023.

[LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, oct 1992.

[LKKO20] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vCNN: Verifiable convolutional neural network. Cryptology ePrint Archive, Report 2020/584, 2020. https://eprint.iacr.org/2020/584.

[LXZ21] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2968–2985. ACM Press, November 2021.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[NKL$^+$23] John J. Nay, David Karamardian, Sarah B. Lawsky, Wenting Tao, Meghana Bhat, Raghav Jain, Aaron Travis Lee, Jonathan H. Choi, and Jungo Kasai. Large language models as tax attorneys: A case study in legal capabilities emergence. https://law.stanford.edu/wp-content/uploads/2023/07/White-Paper_Large-Language-Models-as-Tax-Attorneys.pdf, Jul 2023.

[RBL$^+$21] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. *CoRR*, abs/2112.10752, 2021.

[RIZ$^+$17] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Yi Ding, Aarti Bagul, Curtis P. Langlotz, Katie S. Shpanskaya, Matthew P. Lungren, and Andrew Y. Ng. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *CoRR*, abs/1711.05225, 2017.

[RSR$^+$20] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[SBV+13] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 71–84, New York, NY, USA, 2013. Association for Computing Machinery.

[SHZ+18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. pages 4510–4520, 06 2018.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[TGZ+23] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 71–89, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Tha22] Justin Thaler. Proofs, arguments, and zero-knowledge. *Foundations and Trends® in Privacy and Security*, 4(2–4):117–660, 2022.

[TLI+23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[VSB+22] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap, 2022.

[WJB+17] 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*, CCS '17, page 2071–2086, New York, NY, USA, 2017. Association for Computing Machinery.

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

[ZFZS20] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero knowledge proofs for decision tree predictions and accuracy. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 2039–2053. ACM Press, November 2020.

[ZXZS20] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 859–876, 2020.