Distribution Agreement

In presenting this thesis as a partial fulfillment of the requirements for a degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis in whole or in part in all forms of media, now or hereafter now, including display on the World Wide Web. I understand that I may select some access restrictions as part of the online submission of this thesis. I retain all ownership rights to the copyright of the thesis. I also retain the right to use in future works (such as articles or books) all or part of this thesis.

Xu Han March 26, 2025

Probabilistic Zero-Knowledge Proofs for Neural Network Robustness

By

Xu Han (Andy)

Michelangelo Grigni
Advisor

Department of Computer Science

Michelangelo Grigni
Advisor

Jinho D. Choi
Committee Member

Bree Ettinger
Committee Member

2025

Probabilistic Zero-Knowledge Proofs for Neural Network Robustness

By

Xu Han

Michelangelo Grigni
Advisor

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Department of Computer Science

2025

Abstract

Probabilistic Zero-Knowledge Proofs for Neural Network Robustness
By Xu Han

As machine learning (ML) models are increasingly deployed in high-stakes domains, ensuring their robustness has become critical. However, verifying these properties often requires access to models' secrets, which poses a privacy risk. This thesis proposes a novel cryptographic approach that uses Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARKs) to certify the robustness of neural networks without revealing their internal parameters. Building on prior work such as FairProof and GeoCert, we present a scalable and privacy-preserving method that uses verifiable randomness and circuit-based forward propagation to simulate and prove neural network behavior. Our approach introduces a probabilistic strategy that avoids the computational complexity of exact robustness verification and supports real-world scenarios. Experiments on synthetic multilayer perceptrons demonstrate that our zk-SNARK-based system can distinguish between fair and unfair models while maintaining constant-time verification and manageable proof sizes. This work achieves a practical and scalable method for zero-knowledge verification of machine learning models, enabling secure and privacy-preserving model validation.

Probabilistic Zero-Knowledge Proofs for Neural Network Robustness

By

Xu Han

Michelangelo Grigni
Advisor

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Department of Computer Science

2025

Acknowledgments

I would like to express my deepest gratitude to Dr. Michelangelo Grigni, my advisor, for his invaluable guidance, support, and encouragement throughout the course of this research. I am also sincerely thankful to Dr. Bree Ettinger and Dr. Jinho Choi for being members of my committee and their time.

Additionally, I would like to thank Ivo Kubjas from the Gnark team for his technical assistance, as well as Ph.D Chhavi Yadav for her support on understanding the Fairproof paper.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning is becoming increasingly popular, making it more important than ever to ensure the models used by millions of people across different industries are robust to prevent discrimination. Robustness refers to a model's ability to perform consistently on similar inputs, which is critical to preventing discrimination and ensuring model's reliability[5]. Ensuring robustness comes with significant challenges. Due to privacy concerns, it is often impossible to require companies to publicly release their model weights, which could expose proprietary information or sensitive data. Therefore, it is necessary to develop methods that allow the robustness of machine learning models to be proven while maintaining the confidentiality of their internal structure and preserving the privacy of sensitive information. To address this issue, Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (ZK-SNARK) offer a promising solution. ZK-SNARK is a cryptographic proof that allow one party (the prover) to convince another party (the verifier) that a computation was carried out correctly, without revealing any details about the computation itself.

Building upon the paper *FairProof: Confidential and Certifiable Fairness for Neural Networks*, which calculates the exact fairness properties of neural networks using Zero-Knowledge proofs, this research presented a simpler approach by randomly

sampling adjacent and non-adjacent data-points to simulate the behavior of a multi-layer perceptron [12].

This paper introduces the background and motivation for this thesis, presents the methods used to simulate the real-world scenario, and demonstrates the results of the model. Throughout the research, we have addressed several key challenges, including the randomness in the input generation process, difficulties related to rounding and modular arithmetic in field elements, and the simulation of forward-propagation steps in a multi-layer perceptron. Despite these challenges, we have shown that the proposed model is not only feasible but also capable of offering real-world applications, particularly in ensuring the fairness of machine learning models while maintaining their confidentiality. Furthermore, while the current work demonstrates promising results, there are future studies that can be done to make improvements. These include improving the random number generator, extending the model to more complex neural network architectures, and test with real large language models.

Chapter 2

Background

2.1 zk-SNARK

zk-SNARK (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) are a class of cryptographic proofs that enable a prover to convince a verifier that a computation was executed correctly, without revealing any information about the computation itself. These proofs are both succinct, meaning they are small and efficient to verify, and non-interactive, requiring no back-and-forth communication between the prover and verifier [4]. zk-SNARK guarantee the following critical properties:

- **Completeness:** If the statement is true, the verifier accepts the proof.
- **Soundness:** If the statement is false, no malicious prover can generate a valid proof.
- **Zero-Knowledge:** The proof reveals no information beyond the statement's validity.

zk-SNARK build upon the foundational concept of Zero-Knowledge Proofs (ZKPs), as introduced by Goldwasser [6]. Suppose there is a public circuit Γ that describes the neural-network N, the ZKP allows the owner (prover) of the model convince the

users (verifier) that the model satisfies specific fairness criteria without revealing any information about the model. If there exist an input ω that the output of the model $\mathrm{N}(\omega) = 1$, the prover can construct a proof demonstrating this claim while preserving the confidentiality of ω both and the internal details of N.

Formally, it can be expressed in the following three steps:

1. **Trusted Setup:** A multi-party computation (MPC) protocol is used to generate the proving key (pk) and the verification key (vk) based on the circuit Γ. Formally:

$$(\mathrm{pk}, \mathrm{vk}) = \mathrm{TrustedSetup}(\Gamma)$$

   The keys contain information about the circuit and are used for proof generation and verification.

2. **Proof Generation:** The prover uses the proving key (pk), the public input (x), and the secret input (w) to generate a proof (π). For specific proof generation protocols like Groth16, it is able to achieve constant proof sizes and verification time. Formally:

$$\pi = \mathrm{Prove}(\mathrm{pk}, x, w)$$

3. **Proof Verification:** The verifier uses the verification key (vk), the public input (x), and the proof (π) to check the proof's validity. Formally:

$$\mathrm{Verify}(\mathrm{vk}, x, \pi) \rightarrow \{\mathrm{True}, \mathrm{False}\}$$

2.2 Things Behind the Scenes

In this experiment, we use the Gnark library in Golang. Since Go is a high-level programming language, we do not need to manually compile the circuits, generate the constraints system, and do the trusted setups. However, in order to convince others

that the algorithm is trustable, it is necessary to mention some calculations happened behind the proof generation process [2].

2.2.1 Writing the Circuit

To define a circuit in Gnark, we must describe our computation as a system of constraints. This is done by implementing a circuit structure and defining the constraint relationships. In Gnark, we define the circuit as follows:

```
type MyCircuit struct {
    X1 frontend.Variable
    X2 frontend.Variable
    X3 frontend.Variable
}

func (circuit *MyCircuit) Define(api frontend.API) error {
    api.AssertIsEqual(api.Mul(circuit.X1, circuit.X2), circuit.X3)
    return nil
}
```

This circuit will verify if the product of X1 and X2 equals to X3.

2.2.2 Arithmetic Circuit

Consider the equation:

$$x_1 \cdot x_2 + x_1 + 1 = 22. \tag{2.1}$$

To represent this computation as an arithmetic circuit:

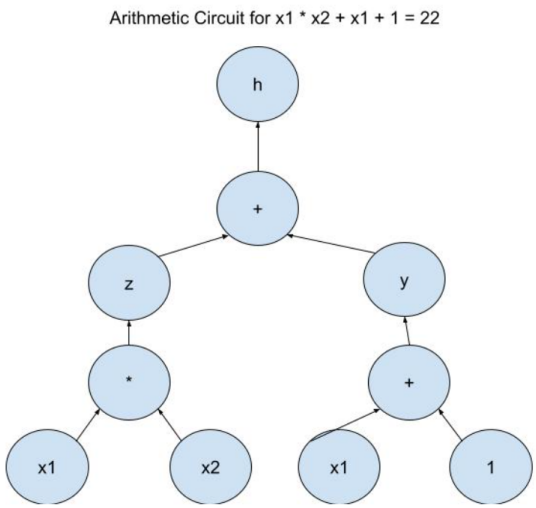

Figure 2.1: Arithmetic Circuit Visualization

1. Introduce an intermediate variable z to store the multiplication result:

$$z = x_1 \cdot x_2. \tag{2.2}$$

2. Express the equation in terms of z:

$$y = (z + x_1) * 1. \tag{2.3}$$

3. Ensure that y satisfies:

$$h = (z + y) * 1. \tag{2.4}$$

Arithmetic circuits are converted into R1CS constraints, allowing efficient verification in zero-knowledge proof systems.

2.2.3 Introduction to Rank-1 Constraint System (R1CS)

The Rank-1 Constraint System (R1CS) is a formalism used to represent arithmetic circuits as a set of constraints. It is widely used in zero-knowledge proof systems such

as zk-SNARK to encode computations in a verifiable manner [1].

An R1CS consists of:

- A set of variables (also known as the witness vector w), which includes both private and public inputs.
- A set of constraints that each computation must satisfy, represented in the form:

$$A \cdot w \odot B \cdot w = C \cdot w. \tag{2.5}$$

- The matrices A, B, and C, which define the constraints applied to the witness vector.

The goal is to ensure that every valid execution of the computation satisfies these constraints while allowing efficient verification.

2.2.4 Example of R1CS

To illustrate how a computation is transformed into R1CS, we continue the previous example equation:

$$x_1 \cdot x_2 + x_1 + 1 = 22. \tag{2.6}$$

Step 1: Define Intermediate Variables

To simplify the computation, we introduce auxiliary variables:

$$z = x_1 \cdot x_2, \tag{2.7}$$

$$y = (1 + x_1) * 1, \tag{2.8}$$

$$h = (z + y) * 1 \tag{2.9}$$

Thus, the witness vector is:

$$w = (1, x_1, x_2, z, y, 22). \tag{2.10}$$

Step 2: Construct the R1CS Constraints

Each step in the computation corresponds to an R1CS constraint:

$$z = x_1 \cdot x_2, \tag{2.11}$$

$$y = (1 + x_1) \cdot 1, \tag{2.12}$$

$$h = (z + y) \cdot 1 \tag{2.13}$$

Step 3: Express in Matrix Form

We now represent the computation in R1CS matrix form:

Constraint	A (Left Side)	B (Right Side)	C (Output Side)
$x_1 \cdot x_2 = z$	(0,1,0,0,0,0)	(0,0,1,0,0,0)	(0,0,0,0,1,0)
$(x_1 + 1) \cdot 1 = y$	(1,1,0,0,0,0)	(1,0,0,0,0,0)	(0,0,0,1,0,0)
$(z + y) \cdot 1 = h$	(0,0,0,1,1,0)	(1,0,0,0,0,0)	(0,0,0,0,0,1)
$\mathbf{w} = (\mathbf{1}, \mathbf{x_1}, \mathbf{x_2}, \mathbf{y}, \mathbf{z}, \mathbf{h})$			

Here is a code of compiling the circuit to generate R1CS:

```
cs, err := frontend.Compile(ecc.BN254.ScalarField(),
                r1cs.NewBuilder, &myCircuit)
```

This R1CS representation allows us to verify the correctness of the computation using cryptographic techniques in zk-SNARK.

2.2.5 Trusted Setup and Groth16 in zk-SNARK

A trusted setup is a critical step in zk-SNARK systems that generates the necessary cryptographic parameters for proof and verification. This setup ensures the security and correctness of the system by establishing parameters that both the prover and verifier will use.

Introduction to Groth16

The Groth16 zk-SNARK scheme is one of the most widely used proving systems due to its efficiency and succinct proofs. It provides constant-size proofs and constant-time verification but requires a trusted setup to generate public parameters.

Groth16 consists of three main phases:

1. Setup Phase: Generates the structured reference string (SRS) containing the proving and verification keys.

2. Proving Phase: Uses the proving key to generate a zk-SNARK proof for a given computation.

3. Verification Phase: Uses the verification key to verify the proof efficiently.

Despite its efficiency, Groth16 requires a separate trusted setup for each unique circuit, which can limit its flexibility.

Purpose of Trusted Setup

The trusted setup is performed to create:

- A proving key (pk), which the prover uses to generate zk-SNARK proofs.

- A verification key (vk), which the verifier uses to check the proof.

The correctness of zk-SNARK highly relies on Groth16 protocol and all components of Groth16 must remain secret to prevent abusing. If the secret is leaked, an attacker could generate fraudulent proofs and falsely convince a verifier [7].

Process of Trusted Setup

The trusted setup process consists of the following steps:

1. **Generating a Random Secret**: A random value, often called the "toxic waste," is generated. This value must be securely discarded after setup.

2. **Computing the Structured Reference String (SRS)**: The SRS contains public parameters derived from the random secret, which are required for proof generation and verification.

3. **Extracting Proving and Verification Keys**: From the SRS, the proving key (pk) and verification key (vk) are extracted.

Trusted Setup in Groth16 Using Gnark

In the Groth16 zk-SNARK scheme, the trusted setup generates public parameters required for proof construction and verification. In the **Gnark** framework, the trusted setup is handled as follows:

1. **Circuit Compilation**: The prover first defines and compiles an arithmetic circuit into an R1CS constraint system.

2. **Parameter Generation**: Gnark runs the setup phase to create the proving and verification keys:

```
pk, vk, err := groth16.Setup(cs)
```

3. **Proving and Verification Keys**: The proving key (pk) is used to generate proofs, while the verification key (vk) is shared with verifiers to check proof validity.

2.3 Robustness of Machine Learning Model

Robustness in machine learning refers to the ability of models to produce reliable and consistent results, ensuring that biases in the data and model inaccuracies do not lead to outcomes that treat individuals unfavorably based on characteristics such as race, gender, or other sensitive attributes [10]. Suppose we have an k-dimension input set S, for a pointwise robustness of a ReLU neural network, the goal is to ensure that small perturbations to any input $x^* \in S$ do not lead to significant changes in the network's output. Mathematically, this can be expressed as:

$$\forall x \in S : \|x - x^*\|_2 \leq \epsilon \implies f(x^*) = f(x) \tag{2.14}$$

where ϵ defines the permissible range of perturbations in the L_2-norm [3].

To verify that a model satisfies robustness properties such as the one defined above, computationally efficient and provably secure methods are needed. This is where cryptographic techniques like Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARK) become valuable. zk-SNARK are a type of cryptographic proof system that allows one party (the prover) to prove the validity of a statement (e.g., a robustness property) to another party (the verifier) without revealing the underlying data or the detailed computations involved .

In the context of machine learning robustness, zk-SNARKs can be used to create verifiable proofs that a model adheres to robustness constraints without exposing sensitive inputs, model parameters, or intermediate computations. For instance, a zk-SNARK can be constructed to certify that for every input x^* within a specified

neighborhood defined by ϵ, the output of the model remains consistent. The key advantage of zk-SNARK is their ability to ensure both privacy and computational efficiency, making them particularly suitable for verifying robustness in applications where data privacy is critical [8].

This combination of machine learning robustness and zk-SNARK is especially relevant in privacy-sensitive domains, such as biomedical applications, where both robustness and data confidentiality are paramount. By integrating zk-SNARK into robustness verification pipelines, users can enhance trust in machine learning systems while ensuring compliance with ethical and regulatory standards [11].

2.4 FairProof and GeoCert

In the paper *FairProof : CONFIDENTIAL AND CERTIFIABLE FAIRNESS FOR NEURAL NETWORKS*, the author utilitize the algorithm inspired by GeoCert to compute the exact pointwise fairness and robustness of neural networks by determining the largest permissible perturbation within an ℓ_p-ball centered around an input [10, 12]. In order to calculate the exact robustness, the naive approach would be calculating the exact euclidean distance between the input point to every decision boundary facet. However, the time complexity of this approach is exponential in hidden neurons, makes it infeasible for large networks. Therefore, GeoCert algorithm is used to simplify the calculation. The *GeoCert* algorithm computes the exact distance from a given point x_0 to the decision boundary of a neural network. It explores the geometric structure of the input space, which is partitioned into polytopes induced by the activation patterns of the network. The algorithm efficiently traverses these polytopes to determine the closest decision boundary.

- **Initialization:** A priority queue Q and a set of visited polytopes C are initialized. The algorithm starts with the polytope $\mathcal{P}(x_0)$ that contains the input point x_0.

: Algorithm 1: GeoCert

Input: point x_0, potential Φ;
Initialization: ;
// Setup priority queue, seen-polytope set;
$Q \leftarrow [\,]; C \leftarrow \{\mathcal{P}(x_0)\}$;
// Handle first polytope's facets;
for *Facet* $\mathcal{F} \in N(\mathcal{P}(x_0))$ **do**
  $Q.push((\Phi(\mathcal{F}), \mathcal{F}))$;
end
// Loop until boundary is popped;
while $Q \neq \emptyset$ **do**
  $\mathcal{F} \leftarrow Q.pop()$;
  if $\mathcal{F}$ *is boundary* **then**
    Return $\mathcal{F}$;
  else
    for $\mathcal{P} \in N(\mathcal{F}) \setminus C$ **do**
      for $\mathcal{F} \in N(P)$ **do**
        $Q.push((\Phi(\mathcal{F}), \mathcal{F})$;
      end
    end
  end
end

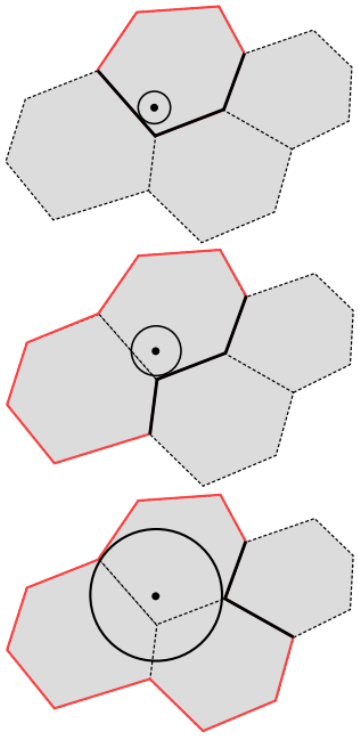

Figure 2.2: GeoCert Computation

- **Processing Initial Facets:** For each facet F of the initial polytope $\mathcal{P}(x_0)$, a priority is computed based on its distance to x_0. These facets are pushed into the priority queue Q.

- **Iterative Search:** It pops the closest facet F from the priority queue, terminates if F lies on the decision boundary, and otherwise explores neighboring unvisited polytopes, adding their facets to the queue with updated priorities.

- **Termination:** The algorithm continues until the decision boundary is found, ensuring an exact computation of the distance.

Those iterations allow the program to check whether the neural network's robustness holds by iteratively searching for the closest decision boundary. If the minimum distance required to reach the boundary is large, the model is considered robust to small adversarial changes. Conversely, if the boundary is close, the model is more

vulnerable to adversarial attacks. Even though this algorithm provides a way to measure the robustness of the machine learning model, it is not as practice due to the following reasons:

- Limitation on ReLU activation function
- Heavy time complexity

Due to its reliance on linear decision boundaries, this approach is restricted to networks with ReLU activations. GeoCert computes the minimum distance to each linear region boundary; however, the number of such boundaries increases exponentially with input dimensionality, resulting in high computational overhead. These reasons make the FairProof algorithm have no feasibility in any of the real-world applications.

2.5 Randomness

A source of randomness is crucial for this experiment in order to convince the verifier that the outputs are not manipulated by the prover. However, to remain non-interactive and prevent sneaky verifiers from getting secret information, it is also not reasonable to let verifier provide the random string. Therefore, having a trusted random string is necessary. Due to the deterministic nature of today's computer, there is no way to let any computer generate a true random number [3]. Therefore, cryptographic protocols rely on alternative methods, such as multi-party computation and trusted setup ceremonies like the Powers of Tau ceremony, to introduce randomness in a verifiable and decentralized manner [9]. For the simplicity purpose, this experiment designed a simple source of verifiable random number generator that generates the random string.

Chapter 3

Approach

While *FairProof* provides a significant advancement in certifying both fairness and robustness of neural networks, it is not without its limitations. One key limitation is its reliance on approximations to compute distances to decision boundaries. This can lead to inaccuracies in certifying robustness and fairness, particularly in networks with complex or highly non-linear decision boundaries. Another limitation arises from the algorithm's focus on multilayer perceptron (MLP) architectures with ReLU activation functions, which restricts its applicability to other types of neural networks, such as convolutional or recurrent neural networks. This hugely restricts the scope of FairProof in real-world scenarios which they mostly require more diverse and complex architectures.

To address these challenges, this chapter introduces a novel method that builds upon the foundation laid by FairProof while overcoming its key limitations. The proposed approach incorporates:

- A more efficient and scalable framework for computing robustness and fairness metrics, reducing computational overhead while maintaining accuracy.
- Generalization to broader classes of neural networks beyond MLPs, including architectures with non-linear activation functions and other specialized layers.

- Integration of cryptographic techniques, such as zk-SNARK, to ensure privacy-preserving robustness verification, enabling secure deployment in sensitive domains.

For the zk-SNARK approach to be valid, it must satisfy three core properties — completeness, soundness, zero-knowledge. To achieve these properties, this experiment builds up a zk-SNARK circuit that checks the forward calculation process of a multi-layer perceptron neural network with randomly generated public inputs and privates inputs committed by companies using cryptographic protocol. The circuit sizes, time for proof, time for verify, proof file size, and verification key file sizes are measured to ensure the scalability and efficiency.

3.1 Public and Private Inputs in the Proof System

In the zk-SNARK proof system, there are two categories of data: **change in some way to detail description of how I actually did it or maybe mention that in latter paragraphs**

- **Public Inputs:** These include the user-provided inputs to the model and the corresponding classification outputs. The public inputs serve as the observable evidence against which robustness and fairness claims are verified. In order to ensure the users cannot engineer the inputs to obtain information from the machine learning model, the inputs are generated through a verifiable way to ensure randomness.

- **Private Inputs:** These include the model's learned parameters—weights, biases, and hidden-layer outputs. Since these contain proprietary information, they are committed through private commitment scheme that allows verification without direct exposure.

The commitments leverage well-established cryptographic methods, ensuring that even though the verifier does not see the raw weights and biases, they can still validate whether the model satisfies robustness and fairness properties under specific adversarial perturbations.

```
type ProveModelCircuit struct {
 Weights  [2][3][3]frontend.Variable ‘gnark:",private"‘
 Biases   [2][3]frontend.Variable    ‘gnark:",private"‘
 Inputs   [10][3]frontend.Variable   ‘gnark:",public"‘
 Expected [10]frontend.Variable      ‘gnark:",private"‘
}
```

The gnark library provides an intuitive interface for defining variables in a zk-SNARK circuit, clearly distinguishing between public and private inputs using struct tags. In the code snippet above, the ProveModelCircuit struct defines the model components and inputs used in the proof system. Variables tagged with gnark:",private"—such as the Weights, Biases, and Expected outputs—are hidden from the verifier and remain confidential throughout the proving and verification process. On the other hand, variables tagged with gnark:",public"—like the Inputs—are visible to the verifier and serve as the publicly observable evidence used to validate the model's behavior.

This explicit separation ensures that sensitive model parameters remain hidden and tells the program to only include the public parameters for the public witness while still allowing the verifier to confirm whether the model produces consistent and fair outputs for the provided inputs. Compare to circom, Gnark's high-level abstraction significantly simplifies the circuit design process and enforces the zero-knowledge guarantees central to zk-SNARK protocols.

3.2 Verifiable Randomness

If users were allowed to arbitrarily select public inputs, they could potentially craft specific queries that, when combined with observed outputs, reveal partial information about the model's internal parameters. This could lead to unintended privacy leaks, undermining the zero-knowledge non-interactive guarantee of our zk-SNARK approach. To prevent users from strategically selecting public inputs in a way that could reveal information about the private model parameters, a "true" randomness string should be used to generate inputs. This source of randomness can be achieved through a trusted third party.

For the simplistic reason of this experiment, a verifiable random number generator (RNG) is used to generate the random seed for public input data generator. This ensures that public inputs are chosen in an unpredictable and unbiased manner, preventing adversaries from reverse-engineering the model. However, in real-world scenario, a more complicated and secure procedure should be taken to better ensure security.

Please refer to code in appendix A.

This code provides a simple demonstration of how a verifiable random number generator (RNG) could operate within the constraints of zk-SNARK-friendly computation. Specifically, it uses a Linear Congruential Generator (LCG) to produce pseudorandom values based on an initial seed. Since zk-SNARK circuits operate over finite fields and do not natively support modular arithmetic in the traditional sense, we implement our own modular reduction using the SmallMod function. This function is called as a custom hint (NewHint) in the gnark library, allowing us to perform the modulus operation during proof generation in a verifiable way. While this approach is simplified for experimental purposes, it illustrates the principle of embedding verifiable randomness within a zk-SNARK-compatible framework. In a production setting, this would be replaced with cryptographically secure and non-manipulable randomness sources.

3.3 Inputs Generation

Input generation is a critical step in the zk-SNARK proof system because the selection of inputs directly influences what aspects of the model are being verified. If the input space is not well-sampled or is biased in any way, a malicious prover could construct a model that passes verification while still being unfair or vulnerable to adversarial perturbations.

Even if we start with a truly random seed, vulnerabilities may still arise. If the prover is allowed to select the seed, they could generate one that favors their model—choosing inputs on which the model performs well while concealing unfair behavior elsewhere. Conversely, if the verifier selects the seed, they could potentially craft inputs designed to extract sensitive information about the model's internal parameters, violating the zero-knowledge property. This creates a fundamental tension: neither party can be fully trusted to generate the seed independently.

The best solution to this problem is to allow limited interaction between the prover and verifier prior to the private commitment phase, enabling both parties to jointly contribute to the randomness seed. By using a protocol such as a commit-and-reveal scheme or a coin-flipping protocol, each party can submit a random value, and the final seed is derived from the combination (e.g., via XOR). This ensures that neither party has full control over the seed, and as long as one party is honest, the resulting seed remains unbiased and unpredictable. This interactive setup preserves fairness while maintaining the zero-knowledge guarantees of the system.

3.4 Real World Scenario Simulation

To better simulate a real-world interaction between a prover and a verifier, we modify the experimental setup used in the FairProof paper. In FairProof, the entire proving and verification process was conducted locally within the same environment, without

explicitly separating the roles of the prover and verifier[12]. In our setup, we extract the proof and verification key files from the prover's environment and transmit them to the verifier, ensuring a more realistic simulation of the protocol in a distributed environment.

```
// Write the verification key to a file
vkF, _ := os.Create(vkKeyFile)
defer vkF.Close()
_, _ = vk.WriteTo(vkF)
// Write the proof to a file
proofF, _ := os.Create(proofFile)
defer proofF.Close()
_, _ = proof.WriteTo(proofF)
```

The code above demonstrates how the verification key (vk) and the proof (proof) are serialized and written to separate files within the prover's environment. These files are then transmitted to the verifier, who can independently use them to verify the correctness of the model's behavior without requiring access to the model weights or intermediate computations. This separation reflects a more realistic client-server or decentralized verification scenario, where the verifier operates independently from the prover and must rely solely on the proof and the verification key to validate the claims.

```
vkF, err = os.Open(vkKeyFile)
 assert.NoError(err)
 defer vkF.Close()
 proofF, err = os.Open(proofFile)
 assert.NoError(err)
```

```
defer proofF.Close()
```

The code above allows any party with access to the verification key and proof files to read the contents sent by the prover. This code retrieves all the necessary artifacts to carry out the verification independently. Once the vk and proof are loaded, the remaining steps of the verification process proceed exactly as they would in a local setting—using the standard zk-SNARK verification process. This code demonstrates that the proving and verifying processes can be fully decoupled, enabling trustless and distributed validation of machine learning model fairness and robustness.

3.5 Forward Propagation

To compute the output of the multi-layer perceptron neural network given a set of weights and biases, we employ the standard forward propagation process. The network consists of multiple layers, where each neuron's activation is computed as a weighted sum of the previous layer's activations, followed by an activation function.

Formally, for each layer l, the activation of neuron i is given by:

$$z_i^{(l)} = \sum_j W_{ij}^{(l)} \cdot a_j^{(l-1)} + b_i^{(l)} \tag{3.1}$$

where:

- $W_{ij}^{(l)}$ represents the weight connecting neuron j in layer $l-1$ to neuron i in layer l,
- $a_j^{(l-1)}$ is the activation from the previous layer,
- $b_i^{(l)}$ is the bias term for the neuron.

In our implementation, this computation is performed within the circuit, iterating through each layer and neuron. The weighted sum is calculated using the `Mul` and

`Add` operations from `gnark`'s `frontend.API`. After computing the sum, we apply a non-linear activation function.

Since zk-SNARK work over finite fields, we carefully handle numerical precision and apply a modular reduction technique (`SmallMod`) to keep values within the appropriate range. Additionally, we use the ReLU activation function, defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{3.2}$$

which ensures non-negative activations and prevents negative values from propagating.

At the final layer, we determine the predicted class by computing the index of the neuron with the highest activation (argmax operation). The circuit then enforces a constraint that this predicted class matches the expected output.

This structured forward propagation ensures that the proof captures the correct inference computation while being efficiently verifiable within the zk-SNARK framework.

Please refer to the code in Appendix A.

The code defines the core logic of the zk-SNARK circuit for forward propagation using the gnark framework. Each value—whether it be an input, weight, bias, or intermediate result—is represented as a frontend.Variable, which allows the gnark compiler to translate these operations into verifiable constraints over a finite field.

The circuit processes each input vector by iteratively applying the forward propagation steps layer-by-layer. For each neuron, it computes the weighted sum of its inputs, adds the corresponding bias, and then scales down the result using a custom modular reduction function (SmallMod) to simulate fixed-point arithmetic. The scaled value is passed through a ReLU activation function, implemented as a constraint that ensures negative values are clipped to zero. The largeBoundary variable is used as a proxy for handling ReLU within the circuit.

After propagating through all layers, the circuit performs an argmax operation on the final layer's outputs to determine the predicted class. This is achieved using api.Select to conditionally update the current maximum value and its index. Finally, the predicted class index is asserted to match the expected label using api.AssertIsEqual, which guarantees that only correct model behavior will produce a valid zk-SNARK proof.

By representing every step of computation through frontend.Variables and field-compatible operations, this code ensures that the entire inference process is verifiable, preserving the privacy of model parameters while confirming model correctness on public inputs. This exemplifies how neural network inference can be securely and efficiently encoded in zero-knowledge.

3.6 Number Preprocessing

One of the major limitations when working with zk-SNARK systems like Gnark is the lack of native support for floating-point arithmetic. Since all values must be represented as elements within a finite field, it is not possible to directly represent decimal numbers. However, in the context of neural networks, weights, biases, and activations are often floating-point values. This mismatch introduces a challenge in accurately simulating forward propagation within the zk-SNARK circuit.

Initially, we considered a method that separates each floating-point number into its integer and fractional components. Although this approach is conceptually correct, it significantly increases circuit complexity and constraint count due to the additional logic required for decomposition and recomposition.

To address this issue more efficiently, we adopted a fixed-point approximation strategy. Specifically, before embedding the values into the circuit, each floating-point number is scaled by a large constant C (e.g., 1000), effectively converting it into an

integer. Inside the circuit, standard operations such as multiplication and addition are performed on these scaled integers. After each multiplication, the result is divided by the same constant C to maintain the original magnitude, and then rounded to the nearest integer using a custom rounding function. This allows us to simulate real-valued arithmetic with reasonable precision while keeping the circuit lightweight and compatible with Gnark's constraint system.

This scaling and rounding method ensures that the forward propagation logic remains accurate enough for practical use, while maintaining the efficiency and verifiability required for zero-knowledge proofs.

Chapter 4

Experiments

To evaluate the feasibility of the zk-SNARK verification implementation, we conducted experiments focusing on computational complexity, proof size, and verification speed. All experiments were performed on a system equipped with an NVIDIA RTX 3090 GPU and an Intel Core i7-12700KF CPU. The primary objectives of these experiments were to investigate the following key questions:

1 Can this algorithm distinguish between fair and unfair models?

2 Is this algorithm feasible in real-world scenario?

4.1 Fairness Verification

To assess the point-wise fairness robustness of a given machine learning model, we sample a set of test inputs from different regions of the input space. For each sampled point, we generate a set of perturbed inputs within an Euclidean distance of 0.1, simulating small, localized changes. The model is then evaluated on each set of perturbations. If the model's output remains consistent across all perturbations of a particular input, we consider it fair at that point. This defines a notion of *point-wise robustness*: rather than assessing global model fairness, we evaluate whether fairness

holds locally around individual data points. In our experiments, we sample 10 such points and observe their robustness behavior. Because our models are small, we use a strict threshold—requiring 100% consistency across perturbations—but this threshold can be relaxed for more complex models or higher-dimensional inputs. Given a machine learning model randomly generated by ChatGPT, we are able to verify its point-wise fairness robustness. For each of the 10 randomly sampled input points, we generate perturbed variants within an Euclidean distance of 0.1 and evaluate the model's output on these perturbations. Our results show that the model is consistent: all 10 input points exhibit locally stable behavior, meaning the model's predictions remain unchanged within their respective neighborhoods. This indicates that the model satisfies point-wise fairness robustness under our evaluation criterion.

4.2 Feasibility Analysis

To measure the feasibility of the algorithm, all tests were executed in the environment specified above. The following metrics were recorded:

- Proof Generation Time: The time taken to generate zk-SNARK proofs.
- Verification Time: The time required to verify a generated proof.
- Number of Constraints: The total number of constraints generated during the zk-SNARK proving process.

The results align with our expectations. As the size of the weights increases, the number of constraints and proof time exhibit a linear relationship with weight size. However, the verification time remains constant and very fast, highlighting the strong feasibility of this approach.

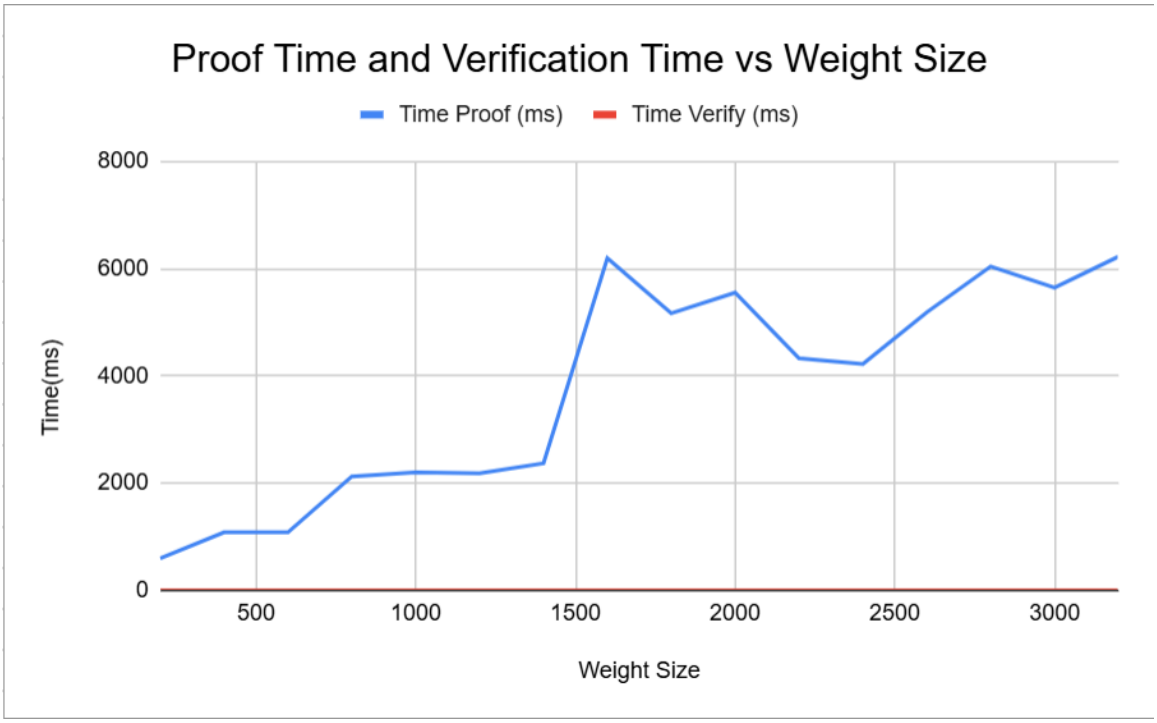

Figure 4.1: Running time for proof and verification on the local machine

Size of weights	Time Proof (ms)	Time Verify (ms)	# Constraints	Weight Size
2*10*10	599.2131	0.4231	628800	200
4*10*10	1082.8269	1.002	1142400	400
6*10*10	1084.5829	0.2189	1656000	600
8*10*10	2126.2322	0.9932	2169600	800
10*10*10	2200.1303	1.0031	2683200	1000
12*10*10	2182.7986	0.5182	3196800	1200
14*10*10	2369.6587	0.1012	3710400	1400
16*10*10	6199.3037	1.0138	4224000	1600
18*10*10	5167.8158	1.5067	4737600	1800
20*10*10	5555.7744	0.4991	5251200	2000
22*10*10	4329.0305	0.1729	5764800	2200
24*10*10	4223.5518	1.0411	6278400	2400
26*10*10	5186.8476	0.4927	6792000	2600
28*10*10	6041.7201	0.9997	7305600	2800
30*10*10	5648.5457	1.0044	7819200	3000
32*10*10	6226.6578	1.011	8332800	3200

Table 4.1: Running Time Data

The results matches our expectation. As the sizes of the weights increases, the number of constraints and proof time has a linear relation to the size of the weights. However, the verification time remains constant. An additional observation from the graph is the "stair-step" pattern in the proof time. Rather than a smooth linear increase, the graph appears to grow in discrete jumps. I hypothesize that this behavior might be due to the internal workings of Gnark. It's possible that proof time increases significantly once the number of constraints surpasses certain thresholds—perhaps linked to internal batching, circuit segmentation, or memory alignment mechanisms within the Gnark backend. The constant verification time is a particularly important aspect of zk-SNARKs, as it ensures that once a proof is generated, the verification can be performed quickly, making it highly efficient.

Chapter 5

Conclusion

The results of our study demonstrate that our proposed method offers a much more efficient algorithm compares to Fairproof to verifiy the robustness of machine learning model while preserving high accuracy. Our experiments have shown that this method is applicable to small sizes of forward-perceptron neural networks.

One of the key advantages of our method is its ability to maintain rapid verification and relatively low runtimes, even as model complexity increases. Even though the proof computation time might be large, we only required to compute the proof once, and it can be repeatedly used with a constant verification time. This ensures its integration into large-scale systems without significant computational overhead. Additionally, our method successfully captures essential properties of the problem space, providing robust and reliable outcomes.

However, there are inherent limitations to our approach. While our experiments validate its applicability, real-world machine learning models are not limited to feedforward perceptron networks with ReLU activation functions. In addition, our model requires the inputs to be continuous numbers, meaning it is not compatible with large language models whose inputs are usually a set of string. The primary reason we did not explore these extensions in this study was to keep the experiment

focused on feasibility rather than broad applicability. In future research we can extend it to more complex architectures and activation functions.

There are several problems that can be addressed in the future. In order to reach cryptographically secure level, we need a truely random seed. This seed can be obtained through hashing the random inputs or from the Groth16 setup phase. There are some solutions to this problem, but due to the scope of this research, those problems are left for future studies.

Overall, our study provides strong evidence that our method is a viable solution for real-world applications, with clear directions for further refinement and expansion.

Appendix A

Source Code

All source code can be found at `https://github.com/2pir2/HonorThesis`.

A.1 Code For Verifiable Randomness:

```
func smallModHint(mod *big.Int, inputs []*big.Int, outputs []*big.Int) error {
 // Computes a % r = b
 // inputs[0] = a -- input
 // inputs[1] = r -- modulus
 // outputs[0] = b -- remainder
 // outputs[1] = (a-b)/r -- quotient
 if len(outputs) != 2 {
  return errors.New("expected 2 outputs")
 }
 if len(inputs) != 2 {
  return errors.New("expected 2 inputs")
 }
 outputs[1].QuoRem(inputs[0], inputs[1], outputs[0])
 return nil
```

```
}

// SmallMod uses the smallModHint to compute the quotient and remainder
func SmallMod(a, r *big.Int) (quo, rem *big.Int) {
 // Initialize output variables
 rem = new(big.Int)
 quo = new(big.Int)

 // Use the hint function to compute modular division
 err := smallModHint(nil, []*big.Int{a, r}, []*big.Int{rem, quo})
 if err != nil {
  panic(fmt.Sprintf("Error in modular computation: %v", err))
 }

 return quo, rem
}
func main() {
 // Linear Congruential Generator (LCG) constants
 a := big.NewInt(1664525)    // Multiplier
 c := big.NewInt(1013904223) // Increment
 m := big.NewInt(100)        // Modulus (2^32)
 seed := big.NewInt(12345)   // Example seed

 // Step 1: Compute the intermediate result temp = a * seed + c
 temp := new(big.Int).Add(new(big.Int).Mul(seed, a), c)

 // Step 2: Use SmallMod to compute the quotient and remainder
```

```
 quo, rem := SmallMod(temp, m)

 // Step 3: Print the results
 fmt.Println("Seed:", seed)
 fmt.Println("Temp:", temp)
 fmt.Println("Quotient:", quo)
 fmt.Println("Remainder:", rem)
}
```

A.2 Code for Forward Propagation and Rounding

```
func (circuit *ProveModelCircuit) Define(api frontend.API) error {
 largeBoundary := frontend.Variable(1000000000)

 // Iterate over each input vector
 for k := 0; k < len(circuit.Inputs); k++ {
  layerOutputs := circuit.Inputs[k] // Start with the input vector

  // Iterate over each layer
  for layer := 0; layer < len(circuit.Weights); layer++ {
   // Create a new slice for the outputs
   newOutputs := make([]frontend.Variable, len(circuit.Weights[layer]))

   // Iterate over each neuron in the layer
   for i := 0; i < len(circuit.Weights[layer]); i++ {
    sum := circuit.Biases[layer][i]
```

```
  // Compute the weighted sum
  for j := 0; j < len(circuit.Weights[layer][i]); j++ {
   tmp := api.Mul(circuit.Weights[layer][i][j], layerOutputs[j])
   sum = api.Add(sum, tmp)
  }

  // Apply the scale-down function
  sum, rem := SmallMod(api, sum, 1000)
  api.Println(sum, "numb", rem)
  // Apply ReLU activation
  newOutputs[i] = applyReLU(api, sum, largeBoundary)
 }

 // Update layerOutputs by iterating through elements
 for i := range newOutputs {
  layerOutputs[i] = newOutputs[i]
 }

 // Print the outputs after the current layer
 api.Println("Outputs after layer", layer, ":", layerOutputs)
}

// Find the argmax in the final layer's output
maxVal := layerOutputs[0]
maxIdx := frontend.Variable(0)
for i := 1; i < len(layerOutputs); i++ {
```

```
    isLess := cmp.IsLess(api, maxVal, layerOutputs[i])
    maxVal = api.Select(isLess, layerOutputs[i], maxVal)
    maxIdx = api.Select(isLess, frontend.Variable(i), maxIdx)
   }

   // Assert the predicted output matches the expected output
   api.AssertIsEqual(maxIdx, circuit.Expected[k])
  }

  return nil
 }
```

Bibliography

[1] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing*, 20(6): 4733–4751, 2023. doi: 10.1109/TDSC.2022.3232813.

[2] Gautam Botrel, Thomas Piellard, Youssef El Housni, Ivo Kubjas, and Arya Tabaie. Consensys/gnark: v0.12.0, January 2025. URL `https://doi.org/10.5281/zenodo.5819104`.

[3] Houssem Ben Braiek and Foutse Khomh. Machine learning robustness: A primer, 2024. URL `https://arxiv.org/abs/2404.00897`.

[4] Thomas Chen, Hui Lu, Teeramet Kunpittaya, and Alan Luo. A review of zk-snarks, 2023. URL `https://arxiv.org/abs/2202.06877`.

[5] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Rich Zemel. Fairness through awareness, 2011. URL `https://arxiv.org/abs/1104.3913`.

[6] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery. ISBN 0897911512. doi: 10.1145/22145.22178. URL `https://doi.org/10.1145/22145.22178`.

[7] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49896-5.

[8] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. Scaling up trustless dnn inference with zero-knowledge proofs, 2022. URL https://arxiv.org/abs/2210.08674.

[9] Valeria Nikolaenko, Sam Ragsdale, Joseph Bonneau, and Dan Boneh. Powers-of-tau to the people: Decentralizing setup ceremonies. In Christina Pöpper and Lejla Batina, editors, *Applied Cryptography and Network Security*, pages 105–134, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-54776-8.

[10] Luca Oneto and Silvia Chiappa. *Fairness in Machine Learning*, page 155–196. Springer International Publishing, 2020. ISBN 9783030438838. doi: 10.1007/978-3-030-43883-8_7. URL http://dx.doi.org/10.1007/978-3-030-43883-8_7.

[11] Tobin South, Alexander Camuto, Shrey Jain, Shayla Nguyen, Robert Mahari, Christian Paquin, Jason Morton, and Alex 'Sandy' Pentland. Verifiable evaluations of machine learning models using zksnarks, 2024. URL https://arxiv.org/abs/2402.02675.

[12] Chhavi Yadav, Amrita Roy Chowdhury, Dan Boneh, and Kamalika Chaudhuri. Fairproof : Confidential and certifiable fairness for neural networks, 2024. URL https://arxiv.org/abs/2402.12572.