> Disclaimer

# Introduction

As machine learning continues its expansion in the global market, the reliability and integrity of its models have become paramount. This is especially true in the context of Machine Learning as a Service (MLaaS), where there's an inherent need to ensure **model authenticity**, which means guaranteeing that the offered model not only matches its description but also operates within accurate parameters while maintaining a degree of privacy. To achieve this, zk-SNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) has garnered significant attention. Its ability to produce short proofs, regardless of the size of the input data, makes it a prime candidate for integration with ML frameworks like EZKL and Daniel Kang's zkml. However, the challenge of translating ML models into circuits optimized for zero-knowledge-proof systems is non-trivial, particularly for complex neural networks.

Consequently, progress has been made in crafting custom circuits using existing architectures, like zk-STARK, Halo2, and Plonk2. Although these custom circuits can accommodate the sophisticated operations of modern ML models, they often fall short of being scalable, generalized solutions. This situation presents developers with a dilemma: selecting the framework that best suits their specific needs.

To address this issue, I'm developing a zkML benchmark. This tool is designed to assist developers in understanding the trade-offs and performance differences among various frameworks. While many frameworks offer their own benchmark sets, making direct comparisons is complex due to the numerous variables that affect performance. My approach focuses on establishing uniform conditions across all frameworks to provide a practical and straightforward comparison.

Unlike the existing benchmark results conducted by the team at EZKL, which focus on traditional machine learning models including Linear Regression, Random Forest Classification, Support Vector Machine (SVM), and Tree Ensemble Regression, our benchmarks analyze selected zkML frameworks with an emphasis on networks that include **deep neural networks** (DNNs) and **convolutional neural networks** (CNNs).

# Benchmark Methodology

The methodology for benchmarking zkML frameworks is meticulously designed to evaluate and compare the performance and capabilities of various zk proof systems for verifiable inference.

## Primary Metrics

- **Accuracy Loss:** Measures how well the inference generated by each frameworks retains the performance of the original, non-private model. Lower accuracy loss is preferable.
- **Memory Usage:** Tracks the peak memory consumption during proof generation, indicating the system's resource demand.
- **Proving Time:** The time required by each framework to generate a proof, essential for gauging the proof system's efficiency. Note: Proof verification time is considered beyond the scope of this analysis.

- **Framework Compatibility:** Assesses each framework's ability to work with various ML model formats and operators.

## Selected Frameworks:

- **EZKL (Halo 2)**
- **ZKML (Halo 2)**
- **circomlib-ml (R1CS Groth16)**
- **opML (Fraud Proof)**

These frameworks were selected based on criteria such as GitHub popularity, the proof system used, and support for different ML model formats. This variety ensures a broad analysis across distinct zk proof systems.

> Note on Orion Exclusion: The proof generation process for Orion, developed by Gizatech, is executed on the Giza platform. Due to this, the memory usage and time cost metrics during proof generation are not directly comparable with those of other frameworks evaluated in this study. As a result, to maintain the integrity and comparability of our benchmarking analysis, Orion's benchmark results will be excluded from the subsequent sections.

## Benchmarking Tasks

Our benchmarking methodology involves tasks on the MNIST dataset for evaluating frameworks under varying complexity levels:

- **MNIST Dataset:**
  - Simplicity of Task: The MNIST dataset, comprising handwritten digits, serves as a benchmark for assessing the basic capabilities of zkML frameworks.
  - Framework Assessment: This task will gauge how each framework manages simple image data in terms of maintaining accuracy and operational efficiency.
  - Parameter Variation: We will test the frameworks on this dataset with an increasing number of parameters and layers, pushing the boundaries of each framework's capacity.

## Benchmarking Process

The benchmarking process for evaluating zkML frameworks is designed to offer a thorough analysis of their performance across various metrics. Below is a detailed outline of the steps involved.

1. **Neural Network Design:**
   - We meticulously craft the structure of networks ranging from 3-layer DNNs to 6-layer CNNs. This is to assess each framework's ability to efficiently translate Multi-Layer Perceptrons (MLPs) into zk circuits, focusing on both the transpilation process and its efficiency.
2. **Uniform Testing Conditions:**
   - Given the diversity in zkML framework compatibilities, with some frameworks exclusively supporting TensorFlow or PyTorch, establishing uniform testing conditions extends beyond merely standardizing the MLP structures. To facilitate direct and fair performance comparisons, a crucial step involves the unification of model representations across

TensorFlow and PyTorch ecosystems. This harmonization ensures that the benchmarks accurately reflect each framework's capabilities under comparable conditions.
    - Recognizing the challenges posed by existing tools, such as the ONNX framework, which often fails to seamlessly convert models between TensorFlow and PyTorch, I have undertaken a manual approach. This entails defining identical neural network architectures within each framework's preferred environment and meticulously transferring weights and biases in a manner that aligns with the computational paradigms of both TensorFlow and PyTorch. For instance, weights in PyTorch are transposed before being applied to the input matrix, a step that necessitates careful handling to preserve the integrity of the model's computational logic. This meticulous process not only facilitates the assessment of proving time and memory usage across frameworks but also enables a more nuanced evaluation of accuracy loss, thereby ensuring that comparisons between zkML frameworks are both fair and meaningful.

3. **Exclusion of Pre-Processing Steps:**
    - Our measurements concentrate exclusively on the proof generation phase, deliberately omitting data pre-processing or trusted-setup steps to maintain a focused evaluation of proof generation efficiency.
4. **Comparative Performance Analysis:**
    - Performance is compared both horizontally and vertically, examining the impact of variations in MLP structure on critical metrics like proof generation time and memory usage within a single framework, as well as comparing the performance of different frameworks on the same MLP structure.
5. **Highlight Differences:**
    - We clearly delineate performance distinctions and capabilities across the various proving systems. This includes comprehensive insights on each system's response to diverse benchmarking tasks and conditions, especially with modified MLP architectures.
6. **Framework-Specific Feature Evaluation:**
    - Unique or specialized features of each framework, such as the accuracy mode in zkML for precision or resource mode for efficiency, are thoroughly evaluated. This assessment aids in understanding how these features influence the framework's overall performance and utility.

Through this detailed benchmarking process, we aim to provide a nuanced understanding of each zkML framework’s capabilities, especially in handling increasingly complex machine learning models. This approach will guide users in selecting the most suitable framework for their specific requirements in the realm of zkML.

# Results

Before delving into the benchmark results, let's enumerate the tested models.

## DNN

The initial testing network is comprised of Deep Neural Networks (DNN), each featuring an input layer followed by two or three fully connected dense layers. The nomenclature for each model is derived from the size of its layers, separated by underscores ('_'). For instance, the model named "784_56_10" signifies an input size of 784 (corresponding to MNIST dataset images, which are 28x28 pixels grayscale images), a subsequent dense layer with 56 units, and finally, an output layer designed to infer 10 distinct classes. Models "196_25_10" and "196_24_14_10" have their input size reduced from 28x28 to 14x14 pixels to evaluate the performance of the selected frameworks on DNNs with varying layers and parameters. Below are the specifics of each network outlined in the subsequent subsections.

### 784_56_10

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 784)]             0

 dense (Dense)               (None, 56)                43960

 dense_1 (Dense)             (None, 10)                570

=================================================================
Total params: 44530 (173.95 KB)
Trainable params: 44530 (173.95 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

This model achieved an accuracy exceeding 97.40% on the MNIST testing dataset, which contains 10,000 images.

### 196_25_10

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 196)]             0

 dense (Dense)               (None, 25)                4925

 dense_1 (Dense)             (None, 10)                260

=================================================================
Total params: 5185 (20.25 KB)
Trainable params: 5185 (20.25 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

This model achieved an accuracy exceeding 95.41% on the MNIST testing dataset

### 196_24_14_10

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 196)]             0

 dense (Dense)               (None, 24)                4728

 dense_1 (Dense)             (None, 14)                350

 dense_2 (Dense)             (None, 10)                150

=================================================================
Total params: 5228 (20.42 KB)
Trainable params: 5228 (20.42 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

This model achieved an accuracy exceeding 95.56% on the MNIST testing dataset

## CNN

Contrary to the DNN model, where the first element denotes the total input size, the naming convention for CNN models begins with a single dimension of the input shape, such as 28 or 14. This is followed by two Conv2D layers, each coupled with an AvgPooling2D layer to reduce spatial dimensions. After flattening, the network may include zero to two dense layers, culminating in an output dense layer designed for classification.

### 28_6_16_10_5

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)       [(None, 28, 28, 1)]       0

 conv2d (Conv2D)            (None, 24, 24, 6)         156

 re_lu (ReLU)               (None, 24, 24, 6)         0

 average_pooling2d (Aver    (None, 12, 12, 6)         0
 agePooling2D)

 conv2d_1 (Conv2D)          (None, 8, 8, 16)          2416
```

```
 re_lu_1 (ReLU)              (None, 8, 8, 16)          0

 average_pooling2d_1 (Aver  (None, 4, 4, 16)          0
 agePooling2D)

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 10)                2570

=================================================================
Total params: 5142 (20.09 KB)
Trainable params: 5142 (20.09 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

This model achieved an accuracy exceeding 98.66% on the MNIST testing dataset, which consists of 10,000 images.

### 14_5_11_80_10_3

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 14, 14, 1)]       0

 conv2d_10 (Conv2D)          (None, 12, 12, 5)         50

 re_lu_6 (ReLU)              (None, 12, 12, 5)         0

 average_pooling2d_10 (Aver  (None, 6, 6, 5)           0
 agePooling2D)

 conv2d_11 (Conv2D)          (None, 4, 4, 11)          506

 re_lu_7 (ReLU)              (None, 4, 4, 11)          0

 average_pooling2d_11 (Aver  (None, 2, 2, 11)          0
 agePooling2D)

 flatten_5 (Flatten)         (None, 44)                0

 dense_15 (Dense)            (None, 80)                3600

 re_lu_8 (ReLU)              (None, 80)                0

 dense_16 (Dense)            (None, 10)                810

=================================================================
Total params: 4966 (19.40 KB)
Trainable params: 4966 (19.40 KB)
```

```
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

This model achieved an accuracy exceeding 97.07 on the MNIST testing dataset

### 28_6_16_120_84_10_5

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 24, 24, 6)         156

 re_lu (ReLU)                (None, 24, 24, 6)         0

 average_pooling2d (Average  (None, 12, 12, 6)         0
 Pooling2D)

 conv2d_1 (Conv2D)           (None, 8, 8, 16)          2416

 re_lu_1 (ReLU)              (None, 8, 8, 16)          0

 average_pooling2d_1 (Avera  (None, 4, 4, 16)          0
 gePooling2D)

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 120)               30840

 re_lu_2 (ReLU)              (None, 120)               0

 dense_1 (Dense)             (None, 84)                10164

 re_lu_3 (ReLU)              (None, 84)                0
                                                                fklfsjfi

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 44426 (173.54 KB)
Trainable params: 44426 (173.54 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

This model achieved an accuracy exceeding 98.72 on the MNIST testing dataset

> For some unknown reasons, using 'accuracy' mode for ezkl to benchmark on model '196_24_14_10' will crash the testing system since it requires more 128 GB memory. Therefore, we omit this testing

> set in our benchmark and will add this later when this bug has been fixed.

### Increasing Layer

![]

### Increasing Parameter

### Changing Layer

## Analysis

### zkML Delemma

### Layer VS Activition Func

### Sacling Factor