

Bionetta: Efficient Client-Side Zero-Knowledge Machine Learning Proving

Technical Report

Rarimo
Info@rarimo.com

Distributed Lab
contact@distributedlab.com

Abstract

In this report, we compare the performance of our Groth16-based zero-knowledge machine learning framework Bionetta to other tools of similar purpose such as ezkl, Lagrange’s deep-prove, or zkml. The results show a significant boost in the proving time for custom-crafted neural networks: they can be proven even on mobile devices, enabling numerous client-side proving applications. While our scheme increases the cost of one-time preprocessing steps, such as circuit compilation and generating trusted setup, our approach is, to the best of our knowledge, the only one that is deployable on the native EVM smart contracts without overwhelming proof size and verification overheads.

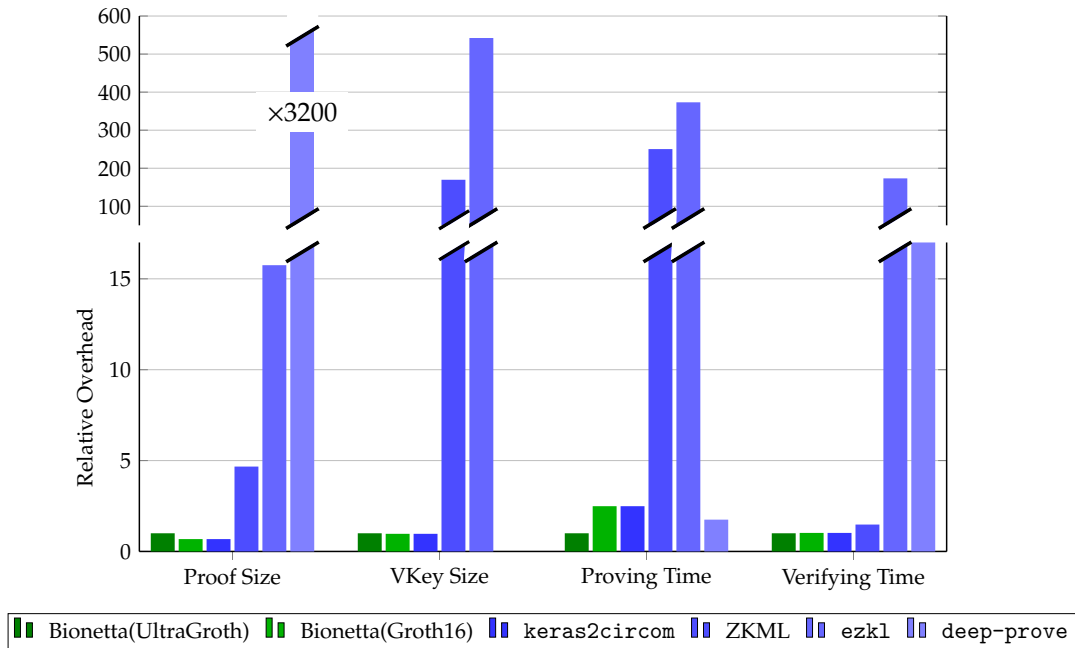


Figure 1: Relative resources overhead of existing zkML frameworks compared to Bionetta (UltraGroth) for a model with ≈ 1 mln parameters. Lower values are better.

Contents

1	Introduction	4
2	Related Approaches	5
3	Proving System	6
3.1	R1CS Arithmetization	6
3.1.1	Proving System Choice	7
3.2	Circuit-Embedded Weights	8
3.2.1	Linear Regression In-Depth Analysis	10
3.3	Float Quantization	11
3.3.1	Quantization Scheme Definition	11
3.3.2	Overflow Handling	12
4	R1CS-Friendly Neural Networks	13
4.1	Activation Functions	13
4.2	Encoder-Decoder Layer	16
4.3	Encoder-Decoder Convolutional Layers	17
4.3.1	Squeeze-and-Excitation Block	18
4.3.2	Encoder-Decoder Convolutional Layer	19
5	UltraGroth: Lookup Tables in R1CS	20
5.1	Quadratic Arithmetic Programs (QAP)	20
5.2	Groth16 Proof Construction	21
5.2.1	Efficiency Analysis	21
5.3	Lookup Tables	22
5.4	UltraGroth Construction	22
5.4.1	Efficiency Analysis for Bionetta	24
6	Experiments	26
6.1	Setup	26
6.2	Results	27

6.2.1	Memory Usage	28
6.2.2	Proving and Verification	28
6.2.3	Compilation Time	29
7	Bar Plot Analysis of Experimental Outcomes	29
7.1	Memory Usage	30
7.2	Proving and Verification	31
7.3	Compilation Time	32
8	Conclusion	32
9	Authors	33
	Appendices	36
A	UltraGroth Security Proof	36
B	Quantization Scheme Proof	40

1 Introduction

Zero-knowledge proof systems have been extensively used in various applications, ranging from blockchain scalability solutions to privacy-preserving identity protocols, such as Rarimo [Mor+19; Rar25]. Currently, the most widely adopted type of zero-knowledge proof system for native Ethereum verifications is zk-SNARKs. In particular, the Groth16 proving system [Gro16] allows a prover to demonstrate the witness knowledge of some statement without revealing the witness itself using just three pairing operations [OREHS24]. The main advantages of zk-SNARKs include:

- ✓ zk-SNARK proofs are *succinct*: the proof size is typically polylogarithmic in the size of the witness, and the verification time is polylogarithmic in the size of the arithmetical circuit. This makes zk-SNARK systems perfect for blockchain applications, where the verification process must be done on-chain and thus should be as efficient as possible.
- ✓ zk-SNARK proofs are *zero-knowledge*: the proof does not reveal any information about the witness corresponding to the statement being proven. While such property might not be crucial for scalability solutions, it is essential for privacy-preserving protocols, which is the primary use case of our research.

Yet, as of today, the majority of zk-SNARK applications are focused on cryptographic primitives, such as proving the knowledge of the pre-image of the hash digest [Gra+21], elliptic curve operations [AEHG22], pairings [Hou23] etc. While these applications are sufficient for most blockchain use cases, many applications still require more complex statements to be proven. One such application is the *machine learning model proof of inference* or, in our particular case, the *liveness check* and *biometric proximity proof*.

In this report, we introduce **Bionetta** — the R1CS-based proving framework with the primary focus on Groth16 derivatives for effective EVM on-chain verification. The main advantages of our approach combined with *UltraGroth* are the following:

- ✓ We inherit all the benefits of Groth16 protocol: constant-sized proof (**160B**), constant verification time (four pairing operations), and small verification key size.
- ✓ The proving time *depends only on the number of non-linearity calls in the neural network*. All *linear operations* (such as matrix multiplications) are **free**.
- ✓ Overall, we claim that *R1CS arithmetization greatly outperforms any PlonKish arithmetization-based approaches* for client-side proving.

However, in contrast to modern ZKML schemes, our methodology is not universal. In particular, we set the following requirements:

- Neural network weights and architecture **must be public** (which is a must for any decentralized identity protocol); we do not support private model weights.
- Currently, we are supporting a limited yet growing set of neural network building blocks and **do not support general onnx models**. In turn, we provide a range of *R1CS-friendly* layers which can be used to construct highly efficient custom neural network architectures and exploit their structure.

2 Related Approaches

Among the existing zkML tools and libraries, we choose the four most notable for benchmarking: `keras2circom` [drC22], `ddkang/zkml` [Che+24; Kan22], `ezkl` [Sou+24], and recent `deep-prove` [Lab25], claiming to boost the proving time by roughly $\times 158^1$. We are also aware of the recent `zkPyTorch` [Xie+25] framework and look forward to testing it in the future, yet currently it is not available for public use (as of June 2025). Below, we provide an informal overview of the differences between our approach and those of the aforementioned open-sourced frameworks.

keras2circom. As the name suggests, `keras2circom` translates the Keras-based models into Circom circuits. While our approach also utilizes Circom and Groth16, `keras2circom` cannot handle deep networks: in fact, as the original repository suggests, on average, the scaling factor of each weight cannot exceed $10^{76/\ell}$, where ℓ is the network depth. Since the number of layers can easily exceed 30, scaling by the factor of roughly $10^{76/30} \approx 340$ would introduce a drastic precision loss. Besides that, `keras2circom` provides much less optimized circuits for the case when model weights are public.

ezkl and ddkang/zkml. Meanwhile, `ezkl` and `zkml` use Halo2 [BGH19; zca25] as their proving system. Since Halo2 relies on the \mathcal{P} loneKish arithmetization instead of R1CS, each addition imposes additional gates to the resulting circuit. For that reason, if the neural network consists of n parameters, L non-linearities and works over the field \mathbb{F} with the bit-size b , the size of the circuit in Halo2 is roughly $O(n)$. In contrast, our approach reduces this complexity down to $O(2^{w+1} + \frac{b}{w} \cdot L)$, where w is the limb size used in the lookup argument (in practice, $w \approx 20$).

Example. Consider the simple fully-connected neural network, inputting $r = 2000$ neurons and outputting $r' = 100$ neurons, governed by equation $f(\mathbf{x}) = \text{ReLU}(W\mathbf{x})$ for some matrix $W \in \mathbb{R}^{r' \times r}$. While Halo2 circuit would roughly consist of $rr' \approx 200000$ gates, the corresponding Circom circuit over BN254 (so that $b = 254$) with $w = 8$ can be reduced down to $2^9 + \frac{254}{8}r' \approx 3700$ constraints, giving $\times 54$ boost over the existing approach.

In addition to smaller circuit sizes, Groth16 provides much better verification key sizes. While the verification key for our tested models takes only 3KB, the verification key of `ddkang/zkml` can reach 650KB and `ezkl` up to 4.2MB.

deep-prove. `deep-prove` [Lab25] is based on a sum-check-based protocol called Ceno [Liu+24]. Although we do confirm that the proving time of `deep-prove` surpasses one provided by `ezkl` (in fact, the prover is linear in contrast to the $O(n \log n)$ approach common in most existing proving systems), the proof sizes are *prohibitively* large: even for small-sized neural networks (see subsequent sections for more details), the proof size can easily reach several megabytes, making `deep-prove` impractical in smart-contract-related applications.

The brief summary of the aforementioned comparison can be seen in Table 1.

¹Based on the original post <https://x.com/lagrangedev/status/1899853995109396618>.

zkML Framework	Provable on mobile device [†]	Effective EVM verification [‡]	Supports generic architectures [‡]	Low numerical error [☆]
Bionetta	✓	✓	✗	✓
keras2circom	✗	✓	✗	✗
ddkang/zkml	✓	✗	✓	✓
EZKL	✗	✗	✓	✓
deep-prove	✓	✗	✓	✗

Table 1: Comparison of capabilities of different zkML frameworks (in the setting of the client-side proving)

[†] 1-million-sized models can be proven in less than 30 seconds with up to 3GB RAM consumption

[‡] Verification procedure is doable on EVM blockchains: reasonable verification and proof sizes (under 10KB), as well as verifying function complexity (no need for recursive wrapping of the proving procedure)

[‡] Copes well with the medium-sized models not specifically designed for zero-knowledge circuits: for example, general .onnx support

[☆] The output of the circuit does not significantly differ from the real output. For example, running the circuits on top of deep-prove results in a significant error while keras2circom can handle only a small number of layers

3 Proving System

In this section, we introduce the main idea behind our approach. We start by recapping what R1CS is [Section 3.1](#), then we proceed to introducing the primary optimization called *circuit-embedded weights* in [Section 3.2](#), then our quantization scheme in [Section 3.3](#), and finally we show how to implement the highly-efficient custom neural network layers in [Section 4](#). Finally, the [Section 5](#) wraps up the discussion by significantly reducing the non-linearity cost from b to $\frac{b}{w}$ constraints.

3.1 R1CS Arithmetization

The R1CS arithmetization is a well-known technique for converting any computational program encoding the NP statement into a set of quadratic checks over the finite field \mathbb{F} , which are then converted to a set of polynomial checks. More formally, any program can be encoded as the following set of quadratic constraints:

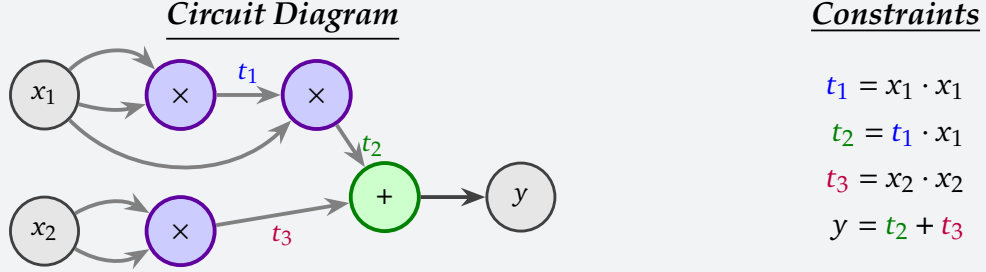
$$\langle \ell_i, \mathbf{z} \rangle \cdot \langle \mathbf{r}_i, \mathbf{z} \rangle = \langle \mathbf{o}_i, \mathbf{z} \rangle, \forall i \in [m],$$

where $\mathbf{z} \in \mathbb{F}^n$ is the witness, roughly speaking representing the intermediate calculations results; $\ell_i, \mathbf{r}_i, \mathbf{o}_i \in \mathbb{F}^n$ are the constant *left*, *right* and *output* vectors, respectively. Finally, we denote the number of constraints by m .

To shorten the notation, vectors $\ell_i, \mathbf{r}_i, \mathbf{o}_i$ can be represented by the constant matrices $L, R, O \in \mathbb{F}^{m \times n}$, respectively. This way, the whole R1CS instance can be simply written

as $Lz \odot Rz = Oz$ and the goal of languages such as Circom [BM+23] is to generate the matrices L, R, O for a given program.

Example. Suppose the program consists in computing the expression $y = x_1^3 + x_2^2$.



In this case, the witness looks as $\mathbf{z} = (1, x_1, x_2, t_1, t_2, t_3, y)^a$. The corresponding matrices are:

$$L = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}, \quad R = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad O = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

^aTo account for a constant term in constraints, we put 1 as the first element of the witness \mathbf{z} .

Now, the main reason R1CS is so attractive for neural network inference is that the R1CS arithmetization is very friendly to the *linear* operations. In particular, consider the following proposition.

Proposition. The main advantage of the R1CS arithmetization is that any addition or multiplication by constants can be done completely for free! Indeed, consider the previous example. Since it contains an addition gate, we can simplify R1CS down to three constraints:

$t_1 = x_1 \cdot x_1$	
$t_2 = t_1 \cdot x_1$	// These two constraints remain the same
$y - t_2 = x_2^2,$	// Combined last two constraints

3.1.1 Proving System Choice

Having the R1CS instance, we can further use a variety of zero-knowledge protocols, including Spartan [Set19], Hyrax [Wah+17], Aurora [BS+18], Fractal [COS19], or WHIR [Arn+24] to generate the proof and the verifier instance (such as the smart-contract). We formally encapsulate this idea in the following definition.

Definition. *Proving System* is a tuple of the following three algorithms:

- $\text{Setup}(1^\lambda, \mathcal{R}) \rightarrow (\text{pp}, \text{vp})$: the setup algorithm takes as input the security parameter $\lambda \in \mathbb{N}$ and the relation \mathcal{R} and outputs the public prover parameters pp and the verifier parameters vp .

- $\text{Prove}(\text{pp}, \mathbb{x}, \mathbb{w}) \rightarrow \pi$: the proving algorithm takes as input the prover parameters pp , the public input \mathbb{x} and the witness \mathbb{w} and outputs the proof π that proves the knowledge of the witness \mathbb{w} for the statement \mathbb{x} such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$.
- $\text{Verify}(\text{vp}, \mathbb{x}, \pi) \rightarrow \{0, 1\}$: the verification algorithm takes as input the verifier parameters vp , the public input \mathbb{x} and the proof π and outputs 1 if the proof π is valid for the statement \mathbb{x} and 0 otherwise.

Remark. Further, we assume that all proving systems in use are *complete* (verification of the valid proof always succeeds), *sound* (verification of an invalid proof fails with overwhelming probability), *zero-knowledge* (the proof does not reveal any information about the witness), *non-interactive*, and, finally, *succinct* (the proof size $|\pi|$ is (poly)logarithmically small in the size of the circuit \mathcal{C} , encoding the relation \mathcal{R}).

In our particular case, we choose the **Groth16** since it is the most widely adopted proving system for zk-SNARKs with the best infrastructure support. Besides, it has the smallest proof and verification key size, so it is a perfect fit for our use case. However, *note that we are not limited to Groth16 and can easily switch to any other proving system*. This is illustrated in the [Figure 2](#). Moreover, to suit our needs, we implemented the custom modification of the Groth16 protocol called *UltraGroth* (see [Section 5](#)) which handles lookup tables, significantly reducing the overhead when proving the execution of non-linear layers.

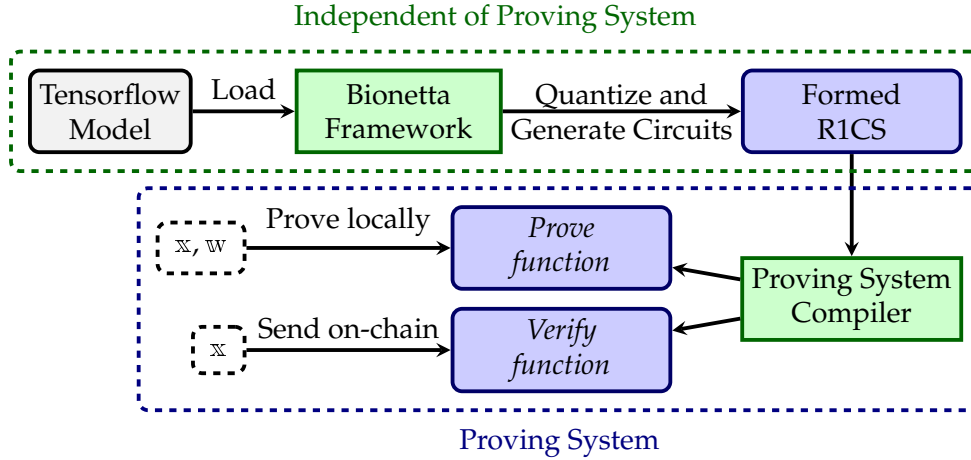


Figure 2: Architecture of the Bionetta framework

3.2 Circuit-Embedded Weights

The previous idea of free linear operations is the basis of our core optimization — *circuit-embedded weights*. To demonstrate its essence, consider the simple example of a **linear regression model**. Such model is governed by the equation:

$$f(\mathbf{x}; \boldsymbol{\theta}) := \langle \boldsymbol{\theta}, \mathbf{x} \rangle + \theta_0, \quad \mathbf{x} \in \mathbb{R}^n, \boldsymbol{\theta} \in \mathbb{R}^{n+1}$$

Currently, the majority of zkML frameworks frames the relation² to be proven as:

$$\mathcal{R}_{\text{ZKML}}^{(f)} = \left\{ \begin{array}{l} \mathbb{x} = (y, \theta) \in \mathbb{R} \times \mathbb{R}^{n+1} \\ \mathbb{w} = (\mathbf{x}) \in \mathbb{R}^n \end{array} \middle| y \approx f(\mathbf{x}; \theta) \right\}$$

Such relation is quite natural: in case we re-train the model (thus, this changes the weights θ), we simply pass the new weights to the prover. However, this approach has a significant drawback: since in such case weights are signals, the prover needs to impose constraints on each term $\theta_i \cdot x_i$ for each $i \in [n]$, even though θ is the part of the public protocol. This means that the prover computes the scalar product $\langle \theta, \mathbf{x} \rangle$ in $O(n)$ constraints. While such complexity is acceptable for small-sized models (especially for such simple cases as the linear regression), when the model becomes deeper, the number of constraints significantly increases with each such linear operation.

In contrast, we propose to treat the weights θ as constants and not as signals. Formally, instead of building the relation $\mathcal{R}_{\text{ZKML}}^{(f)}$ over the public signals θ , we *parameterize* the relation $\mathcal{R}_{\text{Bionetta}}^{(f)}$ over the *constant* weights θ :

$$\mathcal{R}_{\text{Bionetta}}^{(f)}(\theta) = \left\{ \begin{array}{l} \mathbb{x} = (y) \in \mathbb{R} \\ \mathbb{w} = (\mathbf{x}) \in \mathbb{R}^n \end{array} \middle| y \approx f(\mathbf{x}; \theta) \right\}$$

This way, we first train the neural network, then we extract the weights θ , and finally setup the proving system via $\text{Setup}(1^\lambda, \mathcal{R}_{\text{Bionetta}}^{(f)}(\theta))$. In engineering terms, this means that we hardcode the weights θ right into the code. This is illustrated below.

Algorithm 1(a). Pseudocode of the *classical ZKML method* of implementing the linear regression model.

```
circuit LinearRegression(n: int):
    >> public signal input  $\theta[n+1]$ ;
    private signal input  $x[n]$ ;
    public signal output  $y$ ;
     $y <== 0$ ;

    # Scalar product  $\langle \theta, x \rangle$ 
    for i in 1.. $n$ :
         $y += \theta[i] * x[i]$ ;

    # Adding the bias  $\theta_0$ 
     $y <== y + \theta[0]$ ;
```

Algorithm 1(b). Pseudocode of the *Bionetta method* of implementing the linear regression model.

```
# Hardcode the weights
>> const  $\theta[n+1] = [0x642, 0x123, \dots]$ ;

circuit LinearRegression(n: int):
    private signal input  $x[n]$ ;
    public signal output  $y$ ;
     $y <== 0$ ;

    # Scalar product  $\langle \theta, x \rangle$ : now costs 0
    for i in 1.. $n$ :
         $y += \theta[i] * x[i]$ ;

    # Adding the bias  $\theta_0$ 
     $y <== y + \theta[0]$ ;
```

²To further encode the relation into the zero-knowledge circuit, it must operate over the elements from the finite structure (such as finite field \mathbb{F}). For simplicity, our current discussion regards all signals as the real field \mathbb{R} elements, but Section 3.3 will specify how to operate with them over the prime field \mathbb{F}_p .

Proposition. Any matrix-matrix or matrix-vector multiplications in Bionetta where the matrix is fixed **can be implemented in 0 constraints** and the only computational overhead is computing such products over the finite field for a proper witness generation. Consequently, the vast majority of classical machine learning algorithms such as PCA/LDA, Linear/Logistic Regression, linear SVM can be implemented in 0 constraints.

3.2.1 Linear Regression In-Depth Analysis

In this section, we specifically show the difference in both the witness size and the size of R1CS matrices for both approaches. The reader is encouraged to skip this section if he/she is not interested in the low-level details.

With that said, let us write down the L , R , and O matrices for the linear regression model. In case of the classical approach, we need to introduce the witness of total size $3n + 3$. Namely, our witness \mathbf{z} consists of the following elements:

$$\mathbf{z} = (1, \underbrace{\theta_0, \theta_1, \dots, \theta_n}_{\text{Weights } \theta}, \underbrace{x_1, \dots, x_n}_{\text{Inputs } \mathbf{x}}, \underbrace{t_1, \dots, t_n}_{\text{Intermediate variables } \mathbf{t} = \mathbf{x} \odot \theta[1:]}, y) \in \mathbb{F}^{3n+3}$$

Now, we write down $n + 1$ constraints: first n constraints will correspond to the multiplication checks $t_i = x_i \cdot \theta_i$, while the last one will assert $\theta_0 + \sum_{i=1}^n t_i = y$. One can verify that the corresponding checks can be encoded as follows³:

$$\begin{aligned} L &= \begin{bmatrix} \mathbf{0}_n & \mathbf{0}_n & E_{n \times n} & \mathbf{0}_{n \times n} & \mathbf{0}_{n \times n} & \mathbf{0}_n \\ 1 & 0 & \mathbf{0}_n^\top & \mathbf{0}_n^\top & \mathbf{0}_n^\top & 0 \end{bmatrix} \in \mathbb{F}^{(n+1) \times (3n+3)}, \quad // \text{ Encodes left inputs: } L\mathbf{z} = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_n \\ 1 \end{bmatrix} \\ R &= \begin{bmatrix} \mathbf{0}_n & \mathbf{0}_n & \mathbf{0}_{n \times n} & E_{n \times n} & \mathbf{0}_{n \times n} & \mathbf{0}_n \\ 0 & 1 & \mathbf{0}_n^\top & \mathbf{0}_n^\top & \mathbf{1}_n^\top & 0 \end{bmatrix} \in \mathbb{F}^{(n+1) \times (3n+3)}, \quad // \text{ Encodes right inputs: } R\mathbf{z} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \\ \theta_0 + \sum_{i=1}^n t_i \end{bmatrix} \\ O &= \begin{bmatrix} \mathbf{0}_n & \mathbf{0}_n & \mathbf{0}_{n \times n} & \mathbf{0}_{n \times n} & E_{n \times n} & \mathbf{0}_n \\ 0 & 0 & \mathbf{0}_n^\top & \mathbf{0}_n^\top & \mathbf{0}_n^\top & 1 \end{bmatrix} \in \mathbb{F}^{(n+1) \times (3n+3)}, \quad // \text{ Encodes outputs: } O\mathbf{z} = \begin{bmatrix} t_1 \\ \vdots \\ t_n \\ y \end{bmatrix} \end{aligned}$$

However, if we do not pass the weights θ directly into the circuit, we can easily use just a single constraint! Note that our optimized witness \mathbf{z} now consists of $n + 2$ elements:

$$\mathbf{z} = (1, \underbrace{x_1, \dots, x_n}_{\text{Inputs } \mathbf{x}}, y) \in \mathbb{F}^{n+2}$$

So that the corresponding matrices $L(\theta)$, $R(\theta)$, and $O(\theta)$ (notice that now they depend on the weights chosen!) are $L(\theta) \equiv [1, \mathbf{0}_{n+1}^\top]$, $R(\theta) \equiv [0, \theta^\top, 0]$, and $O(\theta) \equiv [\mathbf{0}_{n+1}^\top, 1]$.

³Note that we can reduce this down to n constraints by combining the last two constraints into one. However, this is not crucial for our discussion.

Conclusion. Using the circuit-embedded weights, we can reduce the witness size from $3n + 3$ to $n + 2$ and the number of constraints from $n + 1$ to 1. Similar optimizations can be applied to any other linear operation. For example, the PCA algorithm involves running inference over $f(\mathbf{x}; \mathbf{W}, \boldsymbol{\beta}) = \mathbf{W}\mathbf{x} + \boldsymbol{\beta}$ which is also constraint-free in *Bionetta*.

3.3 Float Quantization

The next section is dedicated to how exactly we encode the real numbers \mathbb{R} arithmetic into the finite field \mathbb{F} arithmetic. The short answer is the following: to encode $x \in \mathbb{R}$, we multiply it by a large power of two, round it, and then convert to the finite field element \mathbb{F}_p . However, the devil is in the details: in such scenario, it is unclear how to handle overflows for deep enough neural networks — the issue that `keras2circom` [drC22], for instance, has not resolved.

3.3.1 Quantization Scheme Definition

Here and hereafter suppose that the finite field \mathbb{F} is prime, that is $\mathbb{F} = \mathbb{F}_p$ for a b -bit prime p . The first question that arises is *how to handle a sign* of some $x \in \mathbb{F}_p$: after all, any element of \mathbb{F}_p must be in range $[0, p)$ and thus cannot be naturally negative. The common approach is to treat the sign of $x \in \mathbb{F}_p$ as follows:

$$\text{sign}(x) = \begin{cases} 1, & \text{if } 0 \leq x < \frac{p-1}{2} \\ -1, & \text{if } \frac{p-1}{2} \leq x < p \end{cases}$$

We propose to simplify such definition a bit: instead of setting the “threshold” for negativity as $\frac{p-1}{2}$, we set it to 2^{b-1} where b is the bitsize of the prime p . This admittedly narrows down the range of possible positive values, yet practically this does not introduce any significant overhead. The primary reason for such choice is (a) convenience, (b) better performance in certain cases.

With such sign handling, it is natural to define the following conversion from the integers \mathbb{Z} to the finite field \mathbb{F}_p (which we call \mathbf{F}_p) and back (we call it \mathbf{Z}):

$$\mathbf{F}_p(x) = \begin{cases} x, & \text{if } x > 0 \\ p + x, & \text{if } x < 0 \end{cases}, \quad \mathbf{Z}(x) = \begin{cases} x, & \text{if } 0 \leq x < 2^{b-1} \\ x - p, & \text{if } 2^{b-1} \leq x < p \end{cases}$$

Now, we are ready to define the quantization process.

Definition. The *Bionetta Quantization Scheme* consists of two algorithms: quantize $Q_\rho : \mathbb{R} \rightarrow \mathbb{F}_p$ and dequantize $D_\rho : \mathbb{F}_p \rightarrow \mathbb{R}$. They are defined as follows:

$$Q_\rho(x) = \mathbf{F}_p(\lfloor 2^\rho x \rfloor), \quad D_\rho(\hat{x}) = 2^{-\rho} \mathbf{Z}(\hat{x})$$

We call $\rho \in \mathbb{N}$ the *precision parameter*. In practice, the larger ρ is, the more precise the quantization is, yet the trickier it is to handle the overflow: see [Section 3.3.2](#).

Example. Suppose we want to multiply $x = 0.2$ and $y = -0.3$ to get $z = xy$ using the Bionetta quantization scheme with a precision $\rho = 5$ and prime field \mathbb{F}_{9967} . The quantized values are:

$$\hat{x} := Q_5(0.2) = \mathbf{F}_p([2^5 \times 0.2]) = \mathbf{F}_p([6.4]) = \mathbf{F}_p(6) = 6$$

$$\hat{y} := Q_5(-0.3) = \mathbf{F}_p([2^5 \times -0.3]) = \mathbf{F}_p([-9.6]) = \mathbf{F}_p(-10) = 9957$$

Then, we multiply quantized values over \mathbb{F}_{9967} and get $\hat{z} = 9907$. Finally, we dequantize the result to get

$$z = D_{10}(9907) = \mathbf{Z}(9907)/2^{10} = -60/1024 \approx -0.059$$

Note that this differs from the real value $z = -0.06$ by a relatively negligible amount.

This example shows an essential observation: when multiplying two quantized values with precisions ρ and ρ' , the dequantization precision must be set to $\rho + \rho'$ (roughly speaking, since $[2^\rho x][2^{\rho'} y] \approx [2^{\rho+\rho'} xy]$). To prove that such dequantization of product indeed leads to the negligible error, we provide the following theorem.

Theorem. Let $\hat{x} := Q_\rho(x)$ and $\hat{y} := Q_{\rho'}(y)$ be the quantized values of x and y , respectively. Then, if we define circuit $C_f(\hat{x}, \hat{y}) = \hat{x} \cdot \hat{y}$ for the multiplication $f(x, y) = x \cdot y$ and assume $\beta := 2 \max\{|x|, |y|\}$ to be relatively small, the following relation for an error ε_ρ holds:

$$\varepsilon_\rho := |D_{2\rho}(C_f(\hat{x}, \hat{y})) - f(x, y)| \leq 2^{-\rho}\beta + 2^{-2\rho}$$

Consequently, the error ε_ρ is negligible in ρ .

We leave the proof in Appendix B for those interested in specifics.

3.3.2 Overflow Handling

The careful reader might notice the primary issue of the quantization scheme: the overflow. Indeed, suppose we want to encode the function $f(x) = x^n$ in the quantized form for fairly large $n \in \mathbb{N}$. The most straightforward way to quantize this expression is to quantize x first (to get $\hat{x} := Q_\rho(x)$), then compute \hat{x}^n and finally dequantize the result with precision $\rho' := n\rho$. While this might be perfectly fine for small n 's (such as 2 or 3), for larger n 's the result might easily overflow the field \mathbb{F}_p .

To handle this issue, we propose the following solution. Suppose we have conducted 4 multiplications in the quantization steps and got the quantized value \hat{x}^4 with precision $4\rho < b - 1$. Then, we can divide the result by $2^{3\rho}$ to get the value of x^4 , but now with precision ρ . This way, we can handle the overflow issue by dividing the intermediate values by $2^{\ell\rho}$ (right shift by $\ell\rho$) for suitable $\ell \in \mathbb{N}$ which we call the **precision cut operation**.

Example. Suppose we have the BN254 prime field \mathbb{F}_p (so that $b = 254$) and we have set the precision parameter $\rho := 80$. We want to compute the value of x^5 for $x = 0.2$. Note that \hat{x}^5

(for $\widehat{x} := Q_\rho(x)$) yields the value with the precision $5\rho = 400$, which is too large for the field \mathbb{F}_p . We propose to build the following circuit $C_f : \mathbb{F}_p \rightarrow \mathbb{F}_p$:

$$C_f(\widehat{x}) = ((\widehat{x}^3 \gg (2\rho)) \times \widehat{x}^2) \gg (2\rho)$$

In this case, $D_\rho(C_f(\widehat{x}))$ would yield the value of x^5 with negligible error. Indeed, $\widehat{x}^3 \gg (2\rho)$ yields x^3 with precision ρ , then multiplication by \widehat{x}^2 rises precision to 3ρ which we finally cut to ρ by the right shift by 2ρ .

Note: $\widetilde{C}_f(\widehat{x}) = (\widehat{x}^3 \gg (2\rho)) \times \widehat{x}^2$ is also a valid circuit, but the dequantization must be done with precision 3ρ so that $D_{3\rho}(\widetilde{C}_f(\widehat{x})) \approx f(x)$.

Now, the core assumption we are making here is that using the precision cut appropriately and the addition/multiplication trick mentioned earlier, we can achieve the negligible error in the quantization process of the whole neural network. Formally, we give the following assumption, which we try to prove.

Assumption. For every neural network $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with the quantization scheme (Q_ρ, D_ρ) , there exists a circuit $C_f : \mathbb{F}^n \rightarrow \mathbb{F}^m$ such that $\|D_\rho(C_f(\widehat{\mathbf{x}})) - f(\mathbf{x})\| = \text{negl}(\rho)$ for any $\|\mathbf{x}\| < \alpha$ for some $\alpha \in \mathbb{R}_{\geq 0}$.

4 R1CS-Friendly Neural Networks

4.1 Activation Functions

Now, with the quantization scheme in mind, we can proceed to the neural network layers that are friendly for the R1CS arithmetization. We already know that the multiplication by matrix $W \in \mathbb{R}^{m \times n}$ and the addition of the bias vector $\beta \in \mathbb{R}^m$ are free in R1CS, so the linear layer $\mathbf{y} = W\mathbf{x} + \beta$ is the most efficient layer in terms of constraints.

However, we cannot simply stack linear layers to form the architecture. Indeed, the composition of linear layers is still a linear layer, so we need to introduce the non-linear activation functions. Unfortunately, many common activations such as **Sigmoid** (corresponding to $\sigma(x) = \frac{1}{1+e^{-x}}$) or **Tanh** (corresponding to $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$) contain exponentials and thus become problematic when it comes to the R1CS. While there are approaches to handle such activations using polynomial approximations [ISY20], we find it more convenient to simply use the activations that can be precisely implemented in R1CS, specified in Table 2. Besides, the large number of modern neural network architectures use such activations as well, so we do not lose much in terms of expressiveness.

The question now is how many constraints do we need to implement, say, **ReLU** or **LeakyReLU** in R1CS.

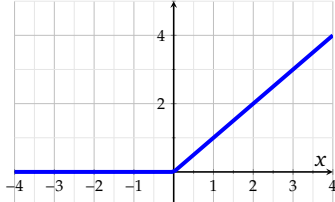
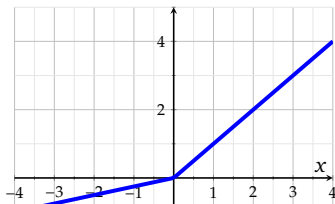
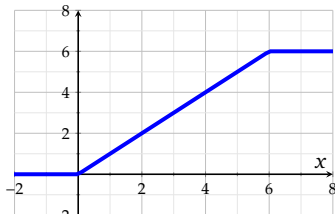
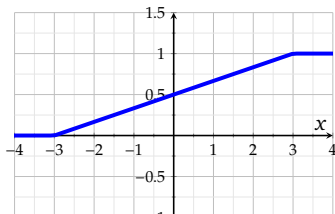
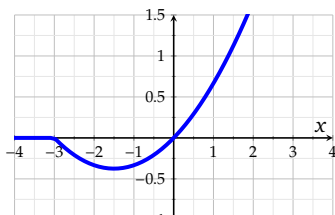
Name	Function	Sketch	#Constraints
ReLU	$\max\{0, x\}$		$\approx b$
LeakyReLU	$\max\{2^{-\ell}x, x\}$		$\approx b$
ReLU6	$\min\{6, \max\{0, x\}\}$		$\approx 2b$
Hard Sigmoid	$\frac{1}{6}\text{ReLU6}(x + 3)$		$\approx 2b$
Hard Swish	$\frac{1}{6}x\text{ReLU6}(x + 3)$		$\approx 2b$

Table 2: The list of activation functions supported by *Bionetta* and their R1CS cost. According to the *MobileNetV3* paper [How+19], we also include effective implementations of sigmoid and swish functions which do not require calculating exponents in-circuit.

Proposition. The ReLU and LeakyReLU activations can be implemented in R1CS with $b + 1$ constraints.

Reasoning. Suppose we are implementing the $\text{ReLU}(x) = \max\{0, x\}$. To compare the field element x with 0, we need to check the $(b - 1)$ th bit. This requires decomposing x into the binary form: $x = \sum_{i=0}^{b-1} x_i 2^i$ where we need to constraint each x_i using quadratic check $x_i(1 - x_i) = 0$. This requires b constraints. Finally, the result is simply $x_b \cdot x$, requiring one more quadratic constraint.

One more interesting feature with ReLU-based activations is that we can *cut the precision for free* (recall that such operation consists in computing $x \gg \rho\ell$).

Proposition. Computing $\text{ReLU}(\hat{x}) \gg \rho\ell$ for some fixed $\ell \in \mathbb{N}$ costs $b + 1$ constraints.

Indeed, since we decompose x into the binary form to check the sign when computing ReLU, we can simply drop the last $\ell\rho$ bits of the decomposed x and get the value of $\text{ReLU}(x)$ with precision cut by $\ell\rho$ for free.

Now, this observation is *crucial*. Indeed, the careful reader could have asked a reasonable question: why don't we use polynomial approximations of ReLU/polynomial activation functions (as deeply researched in Π -nets: see [Chr+20])? The reason is that polynomial approximations cause the significant raise in precision: although computing $Q(\hat{x})$ for d -degree polynomial $Q \in \mathbb{F}[x]$ requires only d constraints per value, such computation increases the precision of \hat{x} by ρd bits. Consequently, even moderately deep networks would require cutting precision of a large number of values, each costing roughly b constraints. This way, the benefit of d constraints per value is lost in the overhead of precision cuts.

This motivates us to use the ReLU-based activations where we can not only compute the activation function with $b + 1$ constraints, but also cut the precision for free. This idea is illustrated in Figure 3.

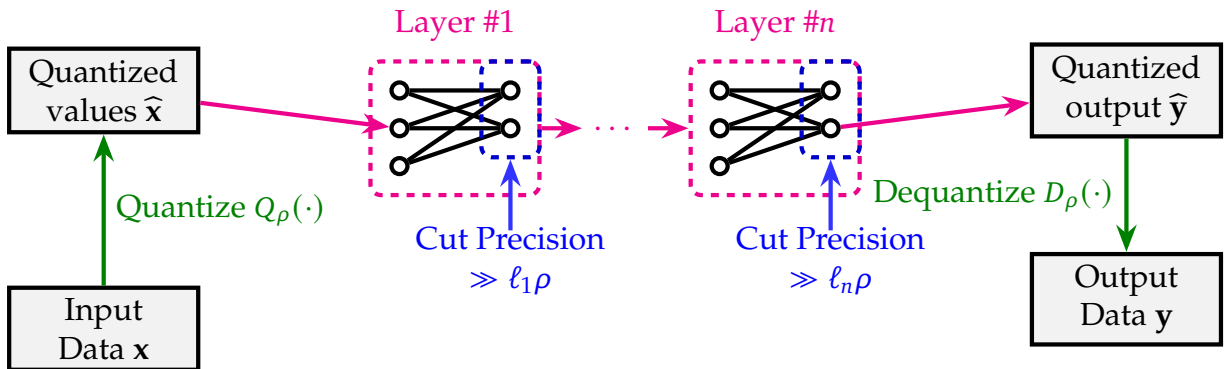


Figure 3: The precision cut operation in the Bionetta system. The dashed magenta boxes represent the layers of the neural network. The dashed blue box represents the output of the layer i , which is cut by $\ell_i\rho$ bits. Finally, green arrows represent the pre and post-processing of the data done outside the circuit.

4.2 Encoder-Decoder Layer

Now, given the fact that the neural network has m non-linear activation calls, the approximate number of constraints for the whole network is roughly mb . Although this does not seem like an issue at first, this quickly becomes one of the biggest problems in constructing the architecture. Consider the fairly simple layer, inputting and outputting $m = 1000$ neurons. Then, $\mathbf{y} = \text{ReLU}(W\mathbf{x})$ would cost $1000b$ constraints, which, for the BN254 field, equates to *roughly 260k constraints*. That is a lot for the single layer.

The primary issue is as follows: assume at some point the output of the layer is $\mathbf{z} \in \mathbb{F}^m$ and we want to apply the activation function ϕ element-wise. This way, we are asserting the mapping $\phi(z_1, \dots, z_m) = [\phi(z_1), \dots, \phi(z_m)]^\top$. Assuming ϕ is ReLU-based activation, we need to *impose mb constraints*. Can we do better?

The answer is *yes*, albeit with certain nuances. Suppose we have a layer with n inputs and m outputs. We inject an additional layer with k hidden neurons and apply the non-linearity ϕ here and remove the non-linearity from the output layer. This way, the encoder-decoder layer would first project the input $\mathbf{x} \in \mathbb{R}^n$ to the latent space $\mathbf{z} \in \mathbb{R}^k$ using the linear (encoder) layer $\mathbf{z} = W_E\mathbf{x}$, then apply the non-linearity ϕ to the latent space \mathbf{z} and finally project the result back to the output space $\mathbf{y} \in \mathbb{R}^m$ using the linear (decoder) layer $\mathbf{y} = W_D\phi(\mathbf{z})$. All in all, we get the forward pass $\mathbf{y} = W_D\phi(W_E\mathbf{x})$.

Definition. The **Encoder-Decoder Layer** is a layer with n inputs and m outputs, which consists of two linear layers (encoder and decoder) and a non-linear activation function ϕ applied to the latent space of the **hidden size** k . It is described as follows^a:

$$\text{EDLayer}(\mathbf{x}; W_D, W_E) = W_D\phi(W_E\mathbf{x})$$

where $W_E \in \mathbb{R}^{k \times n}$ is the encoder matrix, $W_D \in \mathbb{R}^{m \times k}$ is the decoder matrix and ϕ is the non-linear activation function. In case $n = m$, following [He+15], we additionally insert the residual connection between the input and output of the layer, so that the final output is

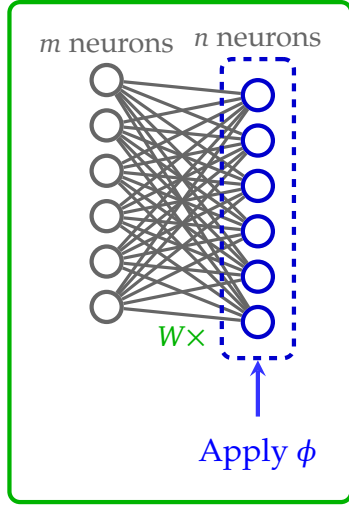
$$\text{EDLayer}(\mathbf{x}; W_D, W_E) = W_D\phi(W_E\mathbf{x}) + \mathbf{x}$$

We call the ratio $\gamma := \frac{k}{m} < 1$ the **squeezing factor** of the ED Layer.

^aWhile formally the forward propagation equation should include the bias term (so that equation becomes $W_D \cdot \phi(W_E\mathbf{x} + \beta_E) + \beta_D$), we omit it for the sake of simplicity by assuming that activation \mathbf{x} is passed with the “1” appended.

Regular Fully-Connected Layer

$$\text{FCLayer}(\mathbf{x}; W) = \phi(W\mathbf{x})$$



Encoder-Decoder Layer

$$\text{EDLayer}(\mathbf{x}; W_E, W_D) = W_D \phi(W_E \mathbf{x}) + \delta_{nm} \mathbf{x}$$

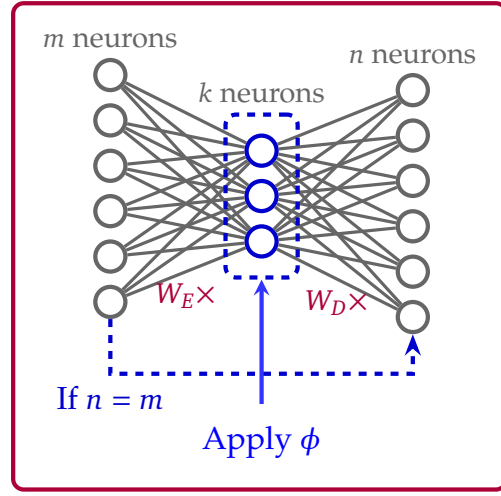


Figure 4: The visual difference between the regular **fully-connected layer** and the **encoder-decoder layer**. The encoder-decoder layer consists of two linear layers (encoder W_E and decoder W_D) and a non-linear activation function ϕ applied to the latent space. The regular fully-connected layer applies the activation function to the output of the layer obtained by the linear projection W .

Of course, this way some accuracy is lost, but we argue that the accuracy loss is insignificant for the proper architecture. Such architecture is mainly beneficial because of the following fact.

Theorem. *The encoder-decoder layer $\text{EDLayer}(\mathbf{x}; D, E)$ can be implemented in R1CS with roughly kb constraints instead of mb, thus giving the speedup of $1/\gamma$ in contrast to the regular fully-connected layer $\text{FCLayer}(\mathbf{x}; W)$.*

4.3 Encoder-Decoder Convolutional Layers

While it is clear how to deal with the fully-connected layers, we need the analogous construction for the convolutional layers, which operate over images. Recall that the goal when working with images is to build the mapping from the volume $\mathbf{X} \in \mathbb{R}^{W \times H \times C}$ (stack of C images of size $W \times H$) to another volume $\mathbf{Y} \in \mathbb{R}^{W' \times H' \times C'}$. This is typically done using the C' kernels of size $f \times f$, which we denote by $\{K_c\}_{c \in [C']} \subseteq \mathbb{R}^{f \times f}$ with stride s which convolves the input image as follows:

$$Y_{i,j,c} = \sum_{k \in [C]} \sum_{u \in [f]} \sum_{v \in [f]} K_{u,v,c} \cdot X_{i:s+u, j:s+v, k}.$$

where the output volume \mathbf{Y} is of size $\frac{W}{s} \times \frac{H}{s} \times C'$. Since such transformation is linear, conducting multiple convolutions would result in a single analogous linear layer.

Therefore, we introduce the non-linearity similarly to the fully-connected layer: simply apply the activation (with the offset) to each output latent variable:

$$Y_{i,j,c} = \phi \left(\sum_{k \in [C]} \sum_{u \in [f]} \sum_{v \in [f]} K_{u,v,c} \cdot X_{i-s+u, j-s+v, k} + \beta_{i,j,c} \right).$$

Now, the primary issue with this version of convolution is that it requires computing roughly $H'W'C'$ non-linear activations (essentially, the whole output volume). To see why that is a problem assume for concreteness that the intermediate volume size turned out to be $W' = H' = C' = 32$, which is not that uncommon to the modern architectures. In that case, for vanilla Groth16, we would require approximately $W'H'C'b \approx 8.36$ mln constraints (with roughly 420 k for the improved UltraGroth protocol). To mitigate this issue, we need to come up with the way to reduce the number of non-linear activations.

4.3.1 Squeeze-and-Excitation Block

One of such examples in the existing literature is the *Squeeze-and-Excitation* (SE) block [HSS18]. The original network proceeds as follows: given the input volume $X \in \mathbb{R}^{W \times H \times C}$, we apply the *global average pooling* operation to get the vector $\mathbf{z} \in \mathbb{R}^C$, where each z_c is the average of the c -th channel of the input volume:

$$z_c = \frac{1}{W \cdot H} \sum_{i=1}^W \sum_{j=1}^H X_{i,j,c}, \quad c \in [C]$$

Next, we apply the gating mechanism to the vector \mathbf{z} to set $\mathbf{s} \in \mathbb{R}^C$, which essentially is the encoder-decoder layer, but with the non-linearity applied at the end: $\mathbf{s} = \sigma(W_D \phi(W_E \mathbf{z}))$. Here, the number of neurons in the hidden layer is $\frac{C}{r}$ for some constant r . Finally, we multiply each channel of the input volume X by the corresponding s_c to get the output volume: $Y[:, :, c] \leftarrow s_c \cdot X[:, :, c]$. The whole computation flow is illustrated in Figure 5.

We give the following proposition to get the number of constraints.

Proposition. *Given that both σ and ϕ cost b constraints to verify, the total cost of the squeeze-and-excitation block is approximately $\left(1 + \frac{1}{r}\right)Cb + HWC$.*

Reasoning. Since all linearities are free, we only need to (a) check the activation over the hidden layer, (b) check the activation over the output layer and (c) check the multiplication of the input volume with the output of the SE block. The first step costs $\frac{C}{r} \cdot b$ constraints, the second step costs Cb constraints and the third step costs HWC constraints. The total cost is therefore $\left(1 + \frac{1}{r}\right)Cb + HWC$.

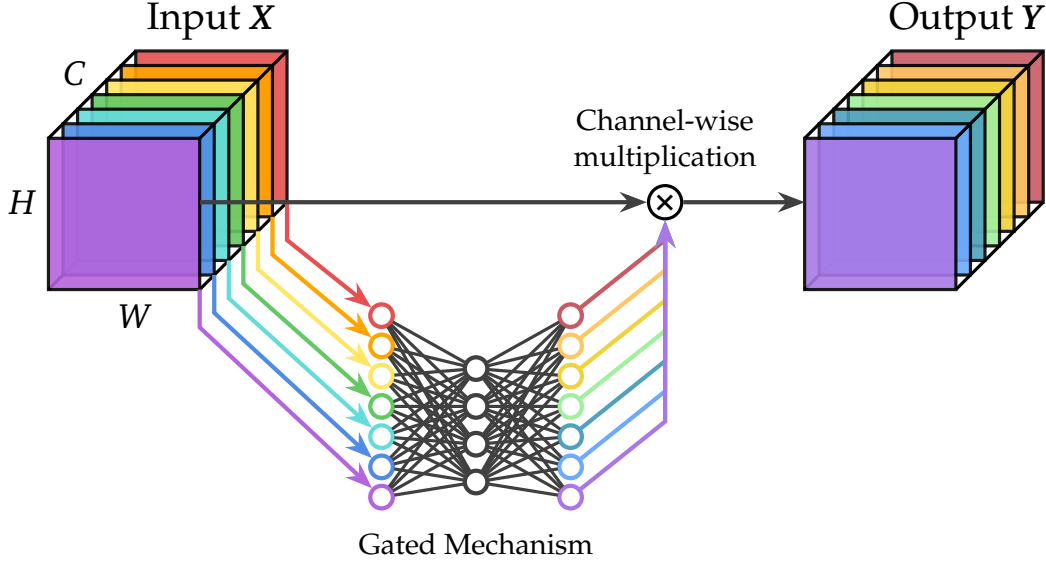


Figure 5: The Squeeze-and-Excitation (SE) block — one of R1CS-friendly layers

One modification of such block is the following: instead of applying the global average pooling along the whole volume, we can effectively split it into blocks (say, forming the grid of size $P_W \times P_H$). This way, we increase the number of constraints by the factor of $P_W P_H$, but the neural network can learn more complex representations.

4.3.2 Encoder-Decoder Convolutional Layer

The issue with the Squeeze-and-Excitation Block is that it does not change the shape of the volume. For that reason, we give the more interesting construction, which is the *encoder-decoder convolutional layer*. We combine certain ideas from the Vision Transformer [Dos+21] and the previously mentioned squeeze-and-excitation block [HSS18].

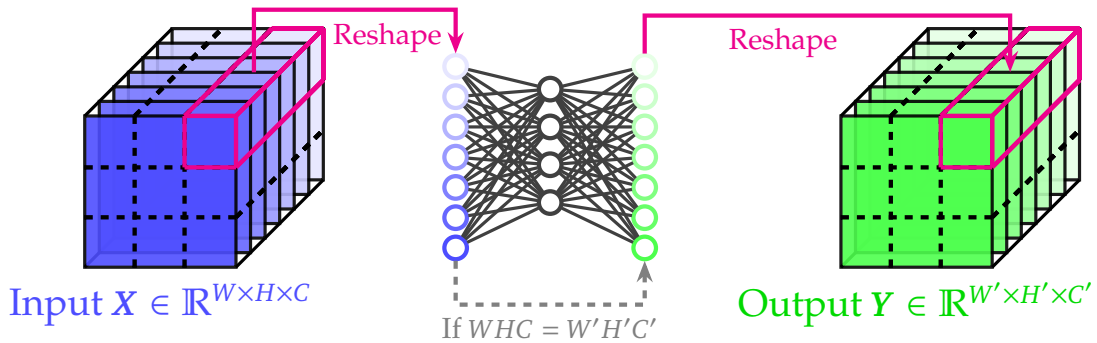


Figure 6: The encoder-decoder convolutional layer. The input volume is split into blocks of size $P \times P \times C$ and the encoder-decoder layer is applied to each block. The output is obtained by combining the outputs of processed blocks with the proper reshaping.

First, we split the input volume into blocks of size $P_W \times P_H$. For simplicity, assume $P_W = P_H = P$ and naturally assume $P \mid W, H$. Assuming the input volume is $X \in \mathbb{R}^{W \times H \times C}$,

we get P^2 blocks $X_{i,j} \in \mathbb{R}^{\frac{W}{P} \times \frac{H}{P} \times C}$ for $i, j \in [P]$. Then, we flatten each block into the vector of size $\frac{WHC}{P^2}$. We connect the flattened blocks to the encoder-decoder layer, each with the K hidden units. Finally, assume we want to get W', H' and C' as the output volume. This way, we connect the latent layer (of size K) to the output layer of size $\frac{W'H'C'}{P^2}$. Finally, we reshape the output volume into the volume of size $\frac{W'}{P} \times \frac{H'}{P} \times C'$ and combine the blocks together to get the final output volume $Y \in \mathbb{R}^{W' \times H' \times C'}$.

Now, let us analyze the cost of such convolution.

Proposition. *Given that the non-linearity in the encoder-decoder layer costs b constraints, the total cost of the encoder-decoder convolutional layer is approximately P^2Kb .*

Reasoning. The only non-linear operation in the encoder-decoder layer is the activation function. Since each processing of encoder-decoder costs Kb constraints and we need to process P^2 blocks, the total cost is P^2Kb .

Note that while the regular convolutional layer costs $W'H'C'b$ constraints, we reduce it significantly to P^2Kb constraints.

5 UltraGroth: Lookup Tables in R1CS

5.1 Quadratic Arithmetic Programs (QAP)

Here we recap the construction of the original Groth16 proving system [Gro16]. From our previous discussion, we know that the R1CS encodes the program in the form $Lz \odot Rz = Oz$ for the given fixed matrices $L, R, O \in \mathbb{F}^{m \times n}$ with $z \in \mathbb{F}^n$. To make the proof succinct, we need to encode this check in the form of a polynomial identity. To do this, we build $3n$ polynomials $\{\ell_i(X)\}_{i \in [n]}, \{r_i(X)\}_{i \in [n]}, \{o_i(X)\}_{i \in [n]} \subseteq \mathbb{F}^{(\leq m)}[X]$ which interpolate the columns of the corresponding matrices:

$$\ell_i(\omega^j) = L_{i,j}, \quad r_i(\omega^j) = R_{i,j}, \quad o_i(\omega^j) = O_{i,j}, \quad i \in [n], j \in [m]$$

Now, the R1CS check can be expressed as the following polynomial identity: the polynomial $m(X) := \sum_{i \in [n]} z_i \ell_i(X) \cdot \sum_{i \in [n]} z_i r_i(X) - \sum_{i \in [n]} z_i o_i(X)$ has roots in the domain $\Omega = \{\omega^j\}_{j \in [m]}$. Equivalently, the vanishing polynomial $t(X) = \prod_{\alpha \in \Omega} (X - \alpha)$ must divide the polynomial $m(X)$. With the suitable choice of Ω (namely, primitive roots of 2^t for suitable t), the expression can be simplified down to $t(X) = X^m - 1$.

All in all, we can define the **Quadratic Arithmetic Program (QAP)** relation as follows:

$$\mathcal{R}_{\text{QAP}} = \left\{ \begin{array}{l} \mathbb{x} = \{z_i\}_{i \in \mathcal{I}_X} \in \mathbb{F}^l \\ \mathbb{w} = \{z_i\}_{i \in \mathcal{I}_W} \in \mathbb{F}^{n-l} \end{array} \middle| \begin{array}{l} \sum_{i \in [n]} z_i \ell_i(X) \cdot \sum_{i \in [n]} z_i r_i(X) = \sum_{i \in [n]} z_i o_i(X) + t(X)h(X) \\ \text{for some polynomial } h(X) \in \mathbb{F}[X] \text{ and } z_0 := 1 \end{array} \right\}$$

Here, l is the number of *public inputs* with indices $\mathcal{I}_X \subseteq [n]$ and n , as previously discussed, is the total solution witness size. We denote the set of indices of the *private inputs* as $\mathcal{I}_W := [n] \setminus \mathcal{I}_X$.

5.2 Groth16 Proof Construction

The Groth16 proving system operates over the bilinear group $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ with the pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. All groups are cyclic and let corresponding generators be g_1, g_2 and g_T , respectively. The field \mathbb{F} is prime: $\mathbb{F} = \mathbb{F}_p$ and the security parameter is $\lambda := p$. Now, we specify the implementation of three key procedures of the proving system: **Setup**(1^λ), **Prove**($\text{pp}, \mathbb{x}, \mathbb{w}$) and **Verify**($\text{pp}, \mathbb{x}, \pi$).

Setup(1^λ). Denote by $\zeta_i(X) := \beta \ell_i(X) + \alpha r_i(X) + o_i(X)$. The trusted setup generates the toxic waste $\alpha, \beta, \gamma, \delta, \tau \xleftarrow{R} \mathbb{F}^\times$ and computes the following common parameters:

$$\begin{aligned} \text{pp} &= \left(g_1^\alpha, g_1^\beta, g_1^\delta, g_2^\beta, g_2^\gamma, g_2^\delta, \left\{ g_1^{\tau^i}, g_2^{\tau^i}, g_1^{\tau^{it(\tau)/\delta}} \right\}_{i \in [n]}, \left\{ g_1^{\zeta_i(\tau)/\gamma} \right\}_{i \in I_X}, \left\{ g_1^{\zeta_i(\tau)/\delta} \right\}_{i \in I_W} \right) \\ \text{vp} &= \left(g_1, g_2, g_2^\gamma, g_2^\delta, g_T^{\alpha\beta}, \left\{ g_1^{\zeta_i(\tau)/\gamma} \right\}_{i \in I_X} \right) \end{aligned}$$

Prove($\text{pp}, \mathbb{x}, \mathbb{w}$). The prover samples random scalars $r, s \xleftarrow{R} \mathbb{F}$ and outputs the proof $\pi = (g_1^{a(\tau)}, g_1^{c(\tau)}, g_2^{b(\tau)})$ where:

$$\begin{aligned} a(X) &= \alpha + \sum_{i \in [n]} z_i \ell_i(X) + r\delta, \quad b(X) = \beta + \sum_{i \in [n]} z_i r_i(X) + s\delta, \\ c(X) &= \frac{1}{\delta} \left(\sum_{i \in I_W} z_i \zeta_i(X) + h(X)t(X) \right) + a(X)s + b(X)r - rs\delta \end{aligned}$$

Verify($\text{pp}, \mathbb{x}, \pi$). The verifier parses the proof $\pi = (\pi_A, \pi_C, \pi_B) \in \mathbb{G}_1^2 \times \mathbb{G}_2$ and accepts the proof if and only if the following check holds:

$$e(\pi_A, \pi_B) = e(g_1^\alpha, g_2^\beta) \cdot e(\pi_{IC}, g_2^\gamma) \cdot e(\pi_C, g_2^\delta).$$

Here, we denote the public input commitment part as $\pi_{IC} := g_1^{\text{IC}(\tau)} \in \mathbb{G}_1$ which the verifier computes where $\text{IC}(X) := \sum_{i \in I_X} z_i \cdot \frac{\zeta_i(X)}{\gamma}$.

5.2.1 Efficiency Analysis

We provide the following proposition.

Proposition. Denote by E_1 and E_2 the cost of exponentiation over \mathbb{G}_1 and \mathbb{G}_2 , respectively. Suppose pairing costs E . Then, the Groth16 proving system complexity is following:

- The proving procedure **Prove**($\text{pp}, \mathbb{x}, \mathbb{w}$) runs in $O(n \cdot E_1 + n \cdot E_2)$.
- The verification procedure **Verify**($\text{vp}, \mathbb{x}, \pi$) requires a time $3E + O(l \cdot E_1)$, while the corresponding verification parameters require $O(l)$ elements from \mathbb{G}_1 .
- The proof π consists of 3 group elements: two from \mathbb{G}_1 and one from \mathbb{G}_2 .

Note that in our case, when the input are private, while the weights are public, the vast majority of signals are private, thus $l \ll n$. For that reason, since verification procedure requires only $O(l)$ elements from \mathbb{G}_1 , the Groth16 verification is effective.

5.3 Lookup Tables

The main idea of the UltraGroth system is to utilize the idea of the lookup tables, which became “native” to PlonKish based proving systems [GW20]. Suppose the values in the lookup tables are $\{t_i\}_{i \in [d]}$ while the part of the witness which we want to check is $\{z_i\}_{i \in [n]}$: that is, we check whether $\{z_i\}_{i \in [n]} \subseteq \{t_i\}_{i \in [d]}$. In our particular case, we will want to have $t_i = i$ for $d = 2^w$ where w is the bit-width of the input. The lookup table will be used to check whether the input is in the range $[0, 2^w)$.

One of the most optimal checks was provided in [Hab22]. We briefly restate the key result of the paper.

Theorem (Following [Hab22]). *Given two sequences of elements $\{t_i\}_{i \in [d]}$ and $\{z_i\}_{i \in [n]}$ from field \mathbb{F} , the inclusion check $\{z_i\}_{i \in [n]} \subseteq \{t_i\}_{i \in [d]}$ is satisfied if and only if there exists the set of multiplicities $\{\mu_i\}_{i \in [d]}$ where $\mu_i = \#\{j \in [n] : z_j = t_i\}$ such that:*

$$\sum_{i \in [n]} \frac{1}{X - z_i} = \sum_{i \in [d]} \frac{\mu_i}{X - t_i}$$

In particular, checking such equality at a random point from \mathbb{F} results in the soundness error of up to $(n + d)/|\mathbb{F}|$, which becomes negligible for large enough $|\mathbb{F}|$.

Note. The second part of the theorem comes from the fact that we can multiply both sides of the equation by the common denominator $\prod_{i \in [n]} (X - t_i) \prod_{i \in [d]} (X - z_i)$. This way, both sides of the equation become polynomials of degree approximately $n + d$. Thus, applying the Schwartz-Zippel lemma, we obtain the soundness of $1 - (n + d)/|\mathbb{F}|$.

Now, the primary issue why this approach is not used in the modern Groth16-based constructions is that there is no efficient way to sample the randomness inside the circuit. Indeed, constructions such as Fiat-Shamir heuristic requires the verifier to hash the transcript, however this has to be done in a “clever way”: for instance, calculating the hash in-circuit would require an overwhelming number of constraints.

5.4 UltraGroth Construction

Suppose besides the regular constraints, there is a need to apply the lookup technique to d subsets of the witness $\{\mathcal{I}_W^{(i)}\}_{i \in [d]} \subseteq \mathcal{I}_W$. We denote by $\mathcal{I}_W^{(d)}$ the remaining part of the witness $\mathcal{I}_W \setminus \bigcup_{i \in [d]} \mathcal{I}_W^{(i)}$. We call such subsets **rounds**: so in total we have $d + 1$ rounds.

Example. *In our particular case, $d = 1$. This way, we have two rounds in total:*

- The round $\mathcal{I}_W^{(0)}$ consists of the outputs of the ReLU function splitted into chunks.
- The round $\mathcal{I}_W^{(1)}$ consists of all remaining checks.

Now, we extend the witness \mathbf{z} to include the indices of sampled randomnesses. Suppose $\{\mathcal{I}_R^{(i)}\}_{i \in [d]}$ are such indices for each round (note that we do not need any randomness in the d^{th} round). We additionally note that in practice each $\mathcal{I}_R^{(i)}$ is commonly a single index with the sampled randomness $\alpha_i \in \mathbb{F}_p$, but to make the protocol more general, assume that $\mathcal{I}_R^{(i)}$ corresponds to the multivariate randomness $\alpha_i \in \mathbb{F}_p^{\ell_i}$ of size ℓ_i . Surely, extending the witness by $\sum_{i \in [d]} \ell_i$ scalars is negligible since this number is typically approximately d (in the particular case of Bionetta, it is simply 1).

In summary, we depict the witness structure in Figure 7.

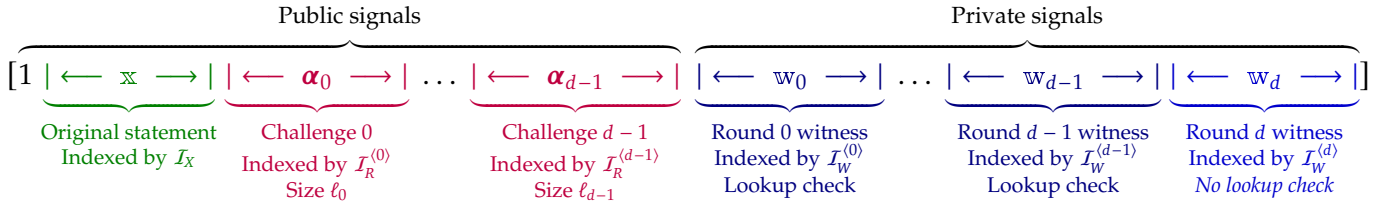


Figure 7: The witness includes three parts: the **public part** \mathcal{I}_X , the **private part** \mathcal{I}_W split into $d+1$ rounds $\{\mathcal{I}_W^{(i)}\}_{i \in [d+1]}$ and the **sampled challenges** $\{\mathcal{I}_R^{(i)}\}_{i \in [d]}$.

Besides the new witness structure, since we are going to apply the Fiat-Shamir heuristic, we use the transformation $\mathcal{H} : \mathbb{F}_p \times \mathbb{G}_1 \rightarrow \mathbb{F}_p$ modeled as a random oracle.

Setup(1^λ). The setup procedure now consists in sampling additional $d+1$ scalars $\delta_0, \dots, \delta_d \xleftarrow{R} \mathbb{F}^\times$. As in the previous section, we denote the polynomial linear combination by $\zeta_i(X) := \beta \ell_i(X) + \alpha r_i(X) + o_i(X)$. The modified common parameters are⁴:

$$\begin{aligned} \text{pp} &= \left(g_1^\alpha, g_1^\beta, \{g_1^{\delta_i}\}_{i \in [d]}, g_2^\beta, g_2^\gamma, \{g_2^{\delta_i}\}_{i \in [d]}, \left\{ g_1^{\tau_i}, g_2^{\tau_i}, g_1^{\tau_i t(\tau)/\delta} \right\}_{i \in [n]}, \left\{ g_1^{\zeta_i(\tau)/\gamma} \right\}_{i \in \mathcal{I}_X}, \left\{ g_1^{\zeta_j(\tau)/\delta_i} \right\}_{(i,j) \in [d] \times \mathcal{I}_W^{(i)}} \right) \\ \text{vp} &= \left(g_1, g_2, g_2^\gamma, g_2^\delta, g_T^{\alpha\beta}, \left\{ g_1^{\zeta_i(\tau)/\gamma} \right\}_{i \in \mathcal{I}_X} \right) \end{aligned}$$

Prove(pp, \mathbf{x} , \mathbf{w}). The prover conducts the steps specified in Algorithm 1.

⁴To simplify the notation, we denote $\{\{a_{i,j}\}_{j \in \mathcal{I}_i}\}_{i \in [n]}$ by $\{a_{i,j}\}_{i,j \in [n] \times \mathcal{I}_i}$.

Algorithm 1: The $\text{Prove}(\text{pp}, \mathbb{x}, \mathbb{w})$ procedure.

```

/* Step 1. Sample challenges for in-circuit lookup check */
1 for round  $i$  in  $[d]$  do
2    $r_i \xleftarrow{R} \mathbb{F}_p$  ; /* Sampling a random scalar */
   /* Compute commitment */
3    $\pi_C^{(i)} \leftarrow g_1^{c_i(\tau)}, c_i(X) := \sum_{j \in \mathcal{I}_W^{(i)}} z_j \frac{\zeta_j(X)}{\delta_i} + r_i \delta_d$ 
4    $a_{i+1} \leftarrow \mathcal{H}(a_i, \pi_C^{(i)})$  ; /* Compute accumulator */
   /* Iteratively fill the  $i^{\text{th}}$  challenge  $\alpha_i \in \mathbb{F}_p^{\ell_i}$  */
5   for  $j \in \mathcal{I}_R^{(i)}$  do
6      $z_j \leftarrow \mathcal{H}(a_{i+1}, g_1^j)$ 
7   end
8 end

/* Step 2. Generate the proof */
9 Compute the polynomial  $h(X)$  as usual from the QAP check
    $\sum_{i \in [n]} z_i \ell_i(X) \cdot \sum_{i \in [n]} z_i r_i(X) = \sum_{i \in [n]} z_i o_i(X) + t(X)h(X)$ 
10 Sample random  $r, s \xleftarrow{R} \mathbb{F}_p$  and compute  $\pi_A \leftarrow g_1^{a(\tau)}, \pi_B \leftarrow g_2^{b(\tau)}$  and
    the last commitment  $\pi_C^{(d)} \leftarrow g_1^{c_d(\tau)}$  where:


$$a(X) = \alpha + \sum_{i \in [n]} z_i \ell_i(X) + r \delta_d,$$


$$b(X) = \beta + \sum_{i \in [n]} z_i r_i(X) + s \delta_d,$$


$$c_d(X) = \frac{1}{\delta_d} \left( \sum_{i \in \mathcal{I}_W^{(d)}} z_i \zeta_i(X) + h(X)t(X) \right) + a(X)s + b(X)r - \sum_{i \in [d]} r_i \delta_i - rs \delta_d$$

11

Output: Proof  $\pi = (\pi_A, \pi_B, \pi_C^{(0)}, \dots, \pi_C^{(d)}) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1^{d+1}$ 

```

$\text{Verify}(\text{pp}, \mathbb{x}, \pi)$. The verifier first recomputes the challenges sampled by the prover $\{\alpha_i\}_{i \in [d]}$ and verifies that the corresponding public signals are correct. Then, it checks:

$$e(\pi_A, \pi_B) = e(g_1^\alpha, g_2^\beta) \cdot e(\pi_{IC}, g_2^\gamma) \cdot \prod_{i \in [d+1]} e(\pi_C^{(i)}, g_2^{\delta_i})$$

5.4.1 Efficiency Analysis for Bionetta

Let us now analyze the efficiency of the UltraGroth system. Consider the Table 3. As can be seen, introducing the lookup check costs only 1 additional pairing and an additional hashing operation. The prover complexity in turn is reduced to $O(2^{w+1} + \frac{bL}{w} + 4L)$ where L is the number of range checks and b is the field bitsize.

Proving System	Prover Complexity	Verifier Complexity	Proof Size
Groth16	$O(bL) E_1$ and E_2	$l E_1, 3 P$	$1 G_2, 2 G_1$
UltraGroth	$O(2^w + \frac{bL}{w}) E_1$ and E_2	$l E_1, (3+d) P, d H$	$1 G_2, (2+d) G_1$
Bionetta	$O(2^w + \frac{bL}{w}) E_1$ and E_2	$l E_1, 4 P, 1 H$	$1 G_2, 3 G_1$

Table 3: The UltraGroth and Bionetta complexity. The prover complexity is measured in terms of the field bitsize b and the number of range checks L . Note that Bionetta is simply the UltraGroth protocol with $d = 1$. Here, E_1 and E_2 are the costs of exponentiation over G_1 and G_2 , respectively. P is the cost of pairing and H is the cost of hashing.

The question, though, is which chunk parameter w to choose. In other words, which value of w minimizes the cost function $c(w; L) = 2^{w+1} + \frac{bL}{w}$ for the given integer L ? The plot itself is visualized in Figure 8. It seems that the best choice would be to pick w around 18 for the given L values. However, we give a more rigorous statement.

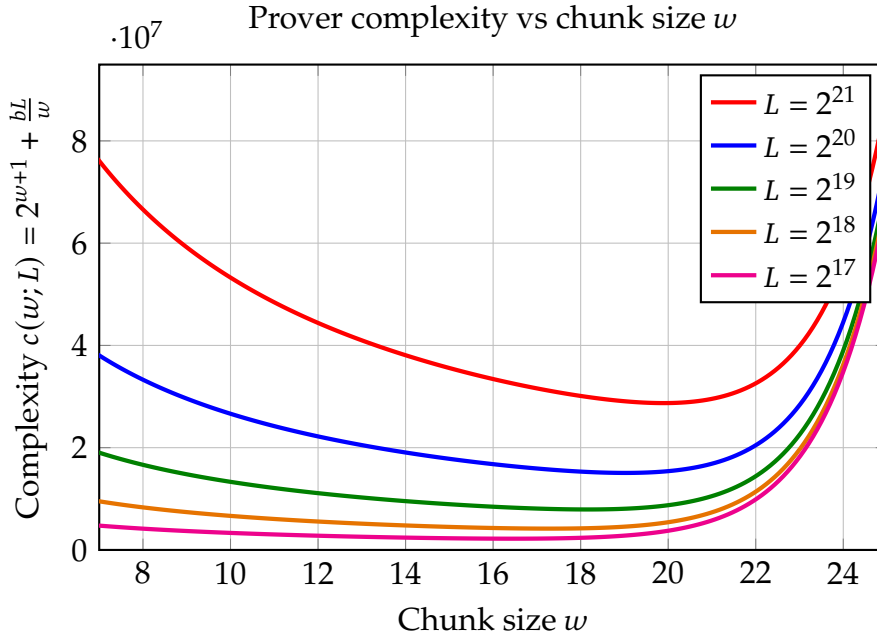


Figure 8: The prover complexity depending on the chunk size w . The different colors represent different values of L .

Proposition. The optimal chunk size \hat{w} must be closest to the solution to the given equation:

$$2^w w^2 = \frac{Lb}{2 \log 2}$$

Reasoning. Simply find the derivative of the function $c(w; L)$ and set it to 0. The derivative is $\partial_w c(w; L) = 2^{w+1} \log 2 - \frac{Lb}{w^2}$. Equating this to zero gives the equation above. It is also easy to check that it is indeed a minimum: notice that the second derivative is positive: $\partial_w^2 c(w; L) = 2^{w+1} \log^2 2 + \frac{2Lb}{w^3}$, thus $\partial_w^2 c(\hat{w}; L) > 0$.

6 Experiments

6.1 Setup

To evaluate the performance of the aforementioned tools, we will check how much time and resources each stage of creating and verifying a proof uses on the specified set of models. To make the comparison fair, we use only standard Dense and ReLU layers and vary the number of linearities (sizes of weights matrices) and the number of non-linearities in the model.

We expect Bionetta to perform better than existing methods when the model has a lot of linear components. On the other hand, models with smaller matrices but many activation functions will likely be harder to prove efficiently on the Bionetta system. We specify the models in [Table 4](#). The [Model 1](#) is minimal, with the only hidden layer to check general overhead and measure the best-case scenario. [Model 2](#) tests the impact of linearities count, [Model 3](#) and [Model 4](#) increase the activations number with little linearities number. Finally, [Model 5](#) and [Model 6](#) combine these factors.

We carried out tests with the latest versions of each library taken from the main branches in the corresponding GitHub repositories [[Sou+24](#); [Kan22](#); [drC22](#); [Lab25](#)].

System Settings. We use TensorFlow 2.12 [[Aba+15](#)] for Bionetta and change TensorFlow/PyTorch version based on the proving system.

Hardware: MacOS, CPU: i5-10400f 32GB RAM;

Software: Bionetta uses Rapidsnark prover for Groth16 and modified Rapidsnark for UltraGroth.

Model 1 Params: 80K Activations: 100	x = Flatten()(inputs) x = Dense(100, activation="relu")(x) outputs = Dense(10)(x)
Model 2 Params: 2M Activations: 10	x = Flatten()(inputs) x = Dense(1000)(x) x = Dense(1000)(x) x = Dense(10, activation="relu")(x) x = Dense(10000)(x) outputs = Dense(10)(x)
Model 3 Params: 50K Activations: 2K	x = Flatten()(inputs) x = Dense(10, activation="relu")(x) x = Dense(1000, activation="relu")(x) x = Dense(10, activation="relu")(x) x = Dense(1000, activation="relu")(x) outputs = Dense(10)(x)
Model 4 Params: 130K Activations: 3K	x = Flatten()(inputs) x = Dense(10, activation="relu")(x) x = Dense(3000, activation="relu")(x) x = Dense(10, activation="relu")(x) x = Dense(3000)(x) outputs = Dense(10)(x)
Model 5 Params: 1M Activations: 2K	x = Flatten()(inputs) x = Dense(1000, activation="relu")(x) x = Dense(100, activation="relu")(x) x = Dense(1000, activation="relu")(x) outputs = Dense(10)(x)
Model 6 Params: 2M Activations: 6K	x = Flatten()(inputs) x = Dense(2000, activation="relu")(x) x = Dense(60, activation="relu")(x) x = Dense(4000, activation="relu")(x) outputs = Dense(10)(x)

Table 4: Model architectures over which we benchmark the existing zkML frameworks. Note that the input (variable inputs) corresponds to the MNIST [Den12] 28×28 image size. Since it is flattened in the first layer, perceive it as a vector of size 784. The output (variable outputs) is the classification of the image into one of the ten classes, so the output layer is a vector of size 10.

6.2 Results

In this section, we compare various stages of proving (such as compilation, proving, or verification time comparison) based on the aforementioned six models. We further visually plot the results and draw the conclusions in Section 7. We highlight the single best result in green while the worst results in red.

6.2.1 Memory Usage

Here, we present the proof, proving key, and verification key sizes for the aforementioned models (Table 5, Table 6, and Table 7, respectively).

Framework	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
Bionetta (UltraGroth)	1.20	1.20	1.20	1.20	1.20	1.20
Bionetta (Groth16)	0.81	0.81	0.81	0.81	0.81	0.81
keras2circom	0.81	0.81	0.81	0.81	0.81	0.81
EZKL	18.9	18.9	18.9	18.9	18.9	18.9
ZKML	5.02	5.60	5.60	5.60	5.60	5.60
deep-prove	1585	5042	3743	3794	4066	4554

Table 5: Proof size, KB.

Framework	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
Bionetta (UltraGroth)	48.4	50.6	80.6	106.3	81.9	156.2
Bionetta (Groth16)	21.3	10.2	396.2	560.2	409.3	1179.6
EZKL	264.1	8454.2	528.4	2114.6	4229.7	8454.2
ZKML	480.2	4226.2	4226.2	4226.2	4226.2	4226.2

Table 6: Proving key size, KB. deep-prove, being a sum-check-based protocol, requires no proving key, so it is excluded from the comparison.

Framework	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
Bionetta (UltraGroth)	3.78	3.79	3.78	3.78	3.78	3.78
Bionetta (Groth16)	3.66	3.66	3.66	3.66	3.65	3.66
EZKL	129.3	4096.5	257.3	1025.5	2049.6	4096.5
ZKML	81.1	641.5	641.5	641.5	641.5	641.5

Table 7: Verification key size, KB. deep-prove, being a sum-check-based protocol, requires no verification key, so it is excluded from the comparison.

6.2.2 Proving and Verification

In this section, we present the proving, witness generation, and verification times for the aforementioned models (Table 8 and Table 9, respectively).

Framework	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
Bionetta (UltraGroth)	0.57	0.73	0.74	1.08	0.89	1.79
Bionetta (Groth16)	0.12	0.27	2.19	2.20	2.22	4.72
EZKL	21.66	656.21	44.88	163.50	332.20	663.93
ZKML	25.00	232.55	209.13	220.84	222.81	230.72
deep-prove	0.378	1.989	0.896	1.010	1.557	2.516

Table 8: Proving time, s

Framework	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
Bionetta (UltraGroth)	0.0056	0.0052	0.0055	0.0057	0.0056	0.0052
Bionetta (Groth16)	0.0056	0.0060	0.0057	0.0056	0.0057	0.0055
EZKL	0.072	1.991	0.138	0.492	0.971	2.324
ZKML	0.0084	0.0112	0.0129	0.0102	0.0083	0.0094
deep-prove	0.034	0.112	0.067	0.079	0.117	0.110

Table 9: Verification time, s.

6.2.3 Compilation Time

Finally, in this section we specify the compilation time for the aforementioned models (Table 10).

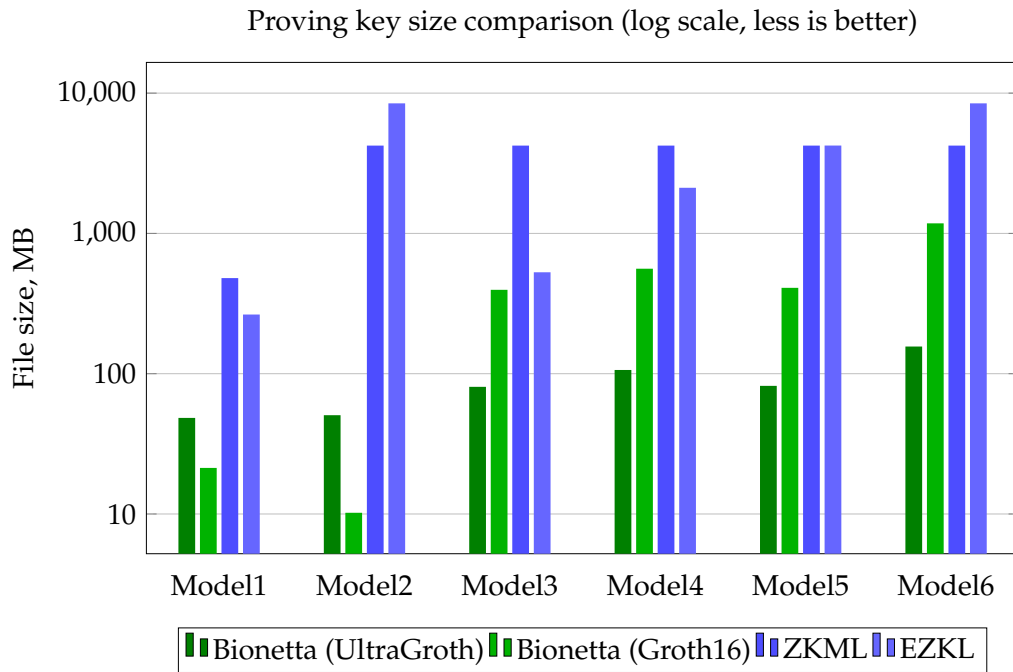
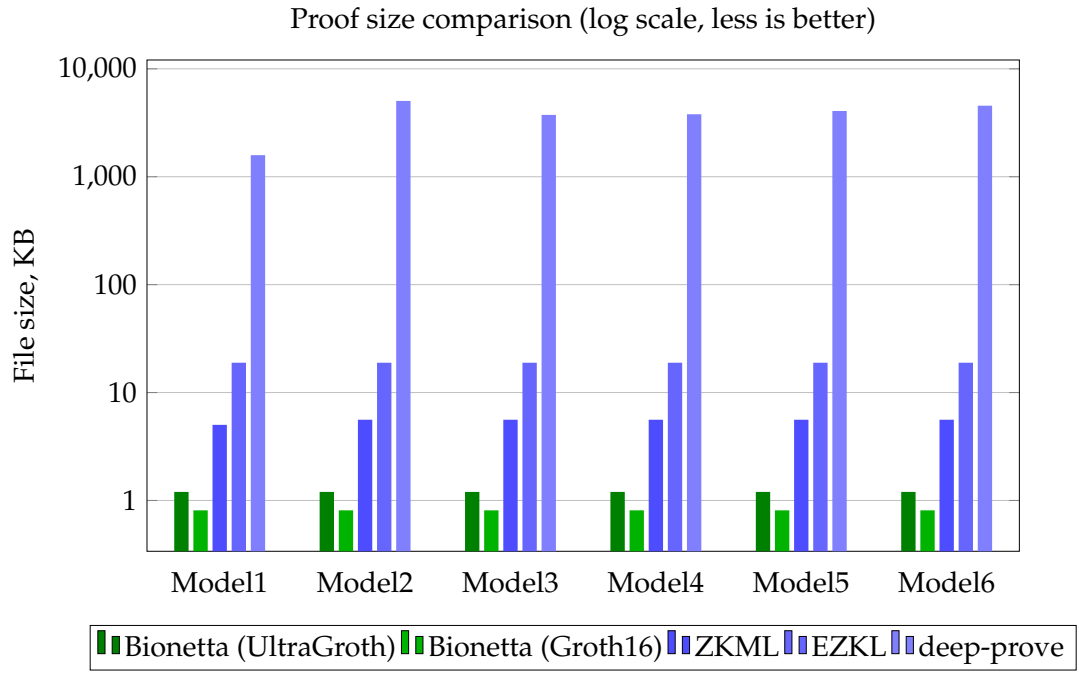
Framework	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
Bionetta (UltraGroth)	512.71	1735.16	856.78	1525.79	1333.76	5413.65
Bionetta (Groth16)	86.34	1347.38	379.15	573.78	750.33	1288.68
EZKL	16.19	503.62	32.40	122.19	245.99	499.39
deep-prove	0.081	1.313	0.036	0.154	0.750	1.357

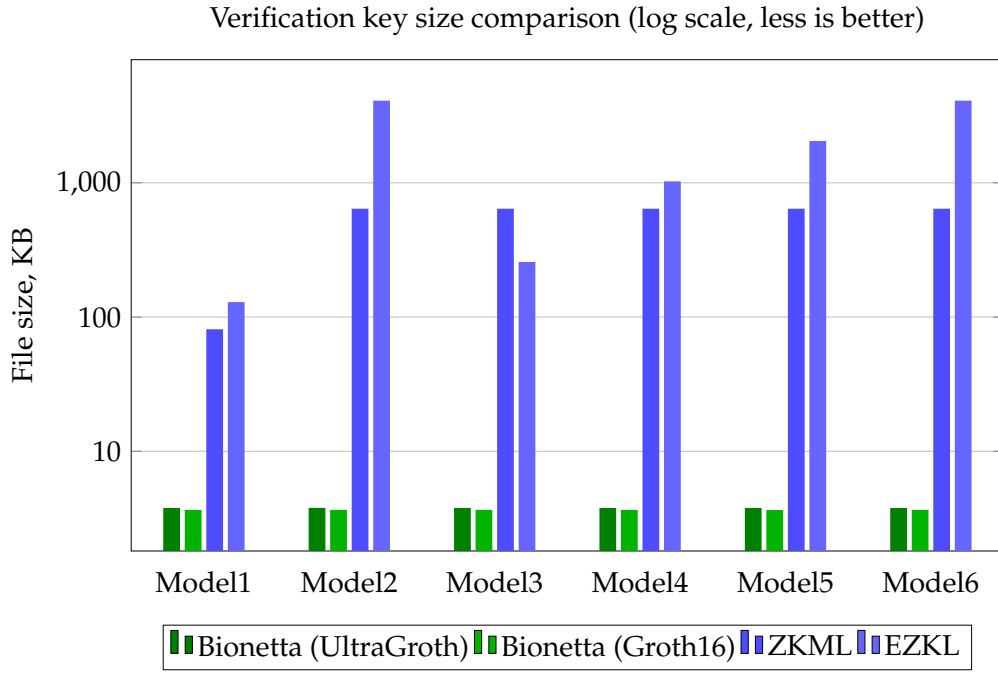
Table 10: Compilation time, s.

7 Bar Plot Analysis of Experimental Outcomes

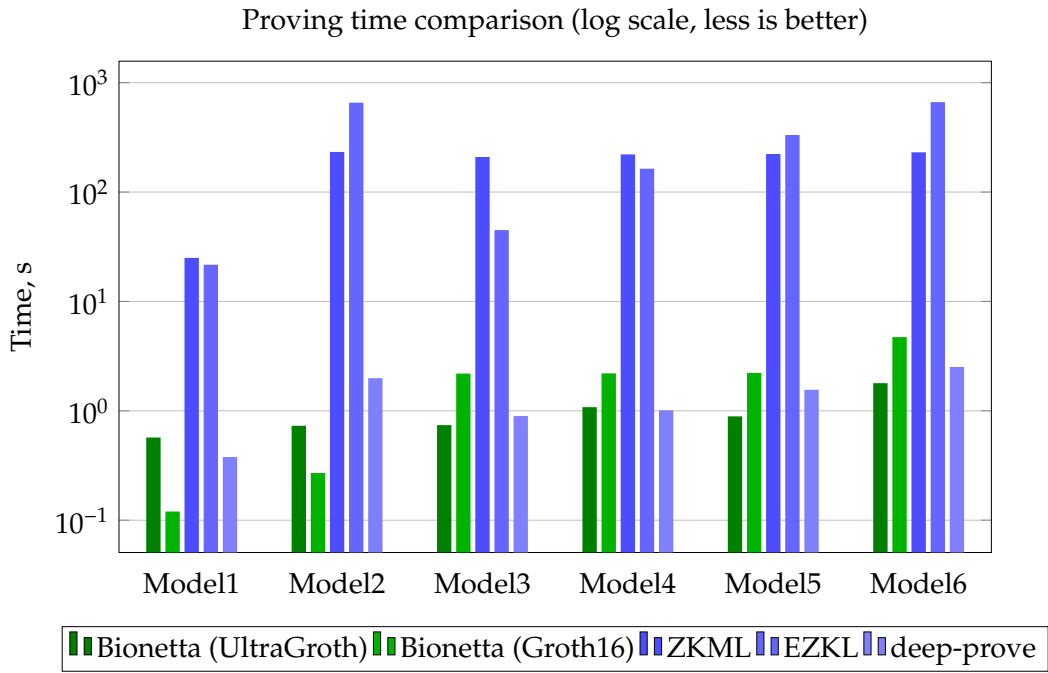
In this section, we visualize the results from the previous section and draw the corresponding conclusions for all proving and verification stages. Please note that we use a logarithmic scale for all the graphs, as the values differ significantly.

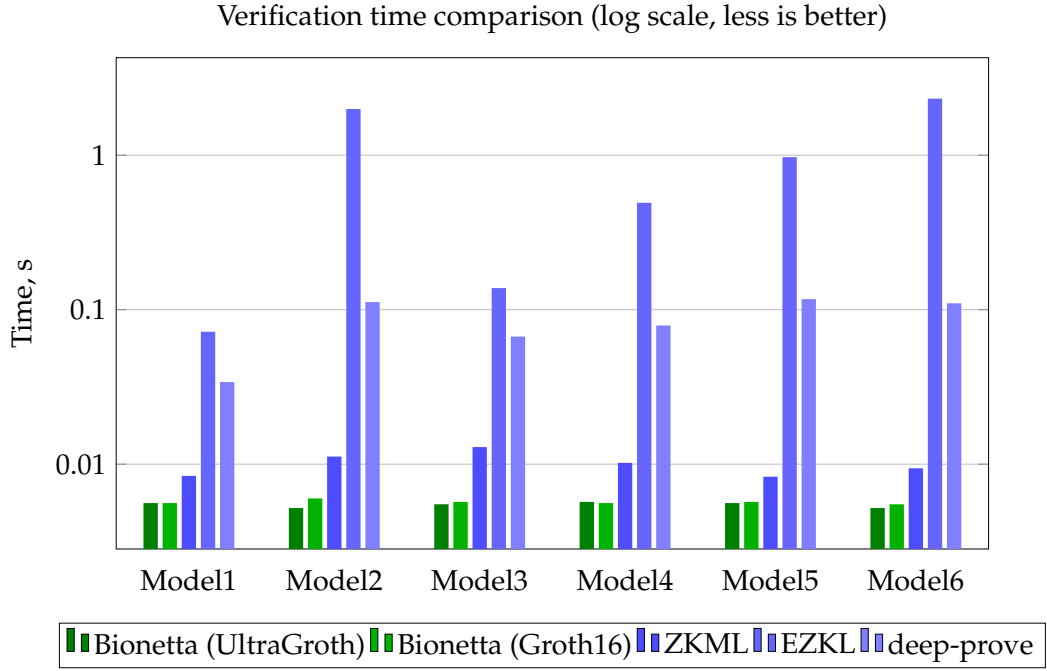
7.1 Memory Usage



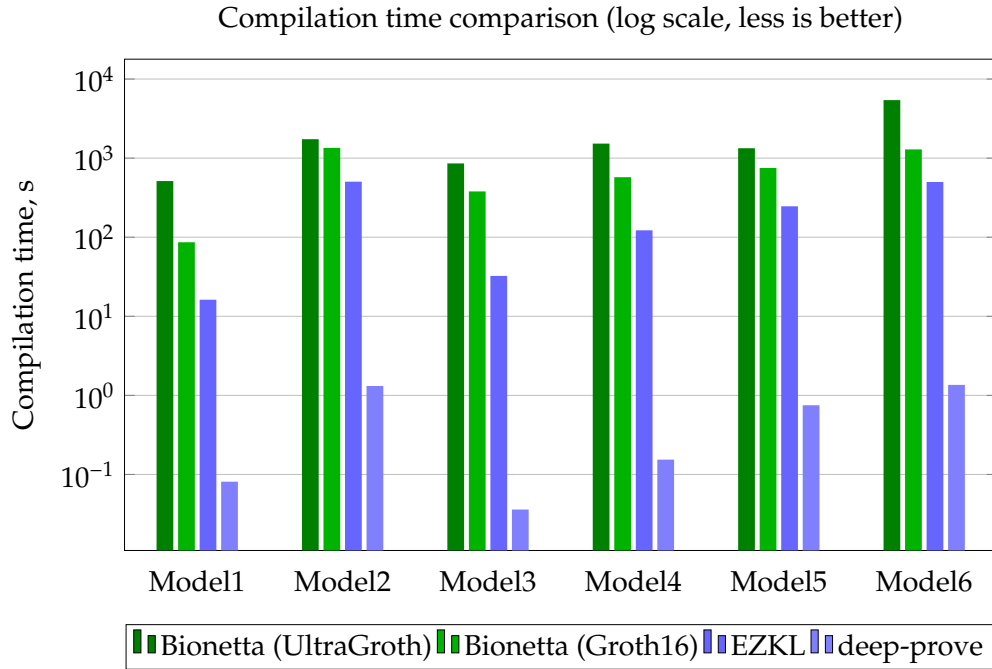


7.2 Proving and Verification





7.3 Compilation Time



8 Conclusion

In this paper, we have presented a new framework for zero-knowledge machine learning, called *Bionetta*. We have experimentally shown that our framework is capable of

proving and verifying machine learning models with high accuracy and low overhead. We have compared our framework with existing frameworks, such as `ddkang/zkml`, `ezkl` and `keras2circom`, and shown that our framework outperforms them in terms of proof size, verification key size, and proving time, while providing a reasonable verification procedure, fully compatible with Ethereum smart contracts.

In addition, we implemented the custom modification of the Groth16 protocol to significantly boost the performance of the proving procedure with minimal overhead for the verification stage. Finally, the Bionetta framework is open-sourced and available for public use, allowing researchers and developers to build upon our work and contribute to the development of zero-knowledge machine learning:

<https://github.com/rarimo/bionetta>

9 Authors

Cryptography

Dmytro Zakharov
Oleksandr Kurbatov
Artem Sdobnov
Yevhenii Sekhin
Vitalii Volovyk

Machine Learning

Mykhailo Velykodnyi
Mark Cherepovskyi
Kyrylo Baibula

Management, Visioning, and Guidance

Lasha Antadze
Pavlo Kravchenko
Volodymyr Dubinin
Yaroslav Panasenko.

References

- [Aba+15] Martín Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from [tensorflow.org](https://www.tensorflow.org). 2015. URL: <https://www.tensorflow.org/>.
- [AEHG22] Diego F. Aranha, Youssef El Housni, and Aurore Guillevic. “A survey of elliptic curves for proof systems”. In: Des. Codes Cryptography 91.11 (Dec. 2022), 3333–3378. ISSN: 0925-1022. DOI: [10.1007/s10623-022-01135-y](https://doi.org/10.1007/s10623-022-01135-y). URL: <https://doi.org/10.1007/s10623-022-01135-y>.
- [Arn+24] Gal Arnon et al. WHIR: Reed–Solomon Proximity Testing with Super-Fast Verification. Cryptology ePrint Archive, Paper 2024/1586. 2024. URL: <https://eprint.iacr.org/2024/1586>.
- [BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive Proof Composition without a Trusted Setup. Cryptology ePrint Archive, Paper 2019/1021. 2019. URL: <https://eprint.iacr.org/2019/1021>.
- [BM+23] Marta Bellés-Muñoz et al. “Circom: A Circuit Description Language for Building Zero-Knowledge Applications”. In: IEEE Transactions on Dependable and Secure Computing 20.6 (2023), pp. 4733–4751. DOI: [10.1109/TDSC.2022.3232813](https://doi.org/10.1109/TDSC.2022.3232813).

- [BS+18] Eli Ben-Sasson et al. Aurora: Transparent Succinct Arguments for R1CS. Cryptology ePrint Archive, Paper 2018/828. 2018. URL: <https://eprint.iacr.org/2018/828>.
- [Che+24] Bing-Jyue Chen et al. “ZKML: An Optimizing System for ML Inference in Zero-Knowledge Proofs”. In: Proceedings of the Nineteenth European Conference on Computer Systems. EuroSys ’24. Athens, Greece: Association for Computing Machinery, 2024, 560–574. ISBN: 9798400704376. DOI: [10.1145/3627703.3650088](https://doi.org/10.1145/3627703.3650088). URL: <https://doi.org/10.1145/3627703.3650088>.
- [Chr+20] Grigorios G Chrysos et al. “P-nets: Deep polynomial neural networks”. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2020, pp. 7325–7335.
- [COS19] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-Quantum and Transparent Succinct Arguments for R1CS. Cryptology ePrint Archive, Paper 2019/1076. 2019. URL: <https://eprint.iacr.org/2019/1076>.
- [Den12] Li Deng. “The mnist database of handwritten digit images for machine learning research”. In: IEEE Signal Processing Magazine 29.6 (2012), pp. 141–142.
- [Dos+21] Alexey Dosovitskiy et al. An Image is Worth 16x16 Words: Transformers for Image Recognition. 2021. arXiv: [2010.11929 \[cs.CV\]](https://arxiv.org/abs/2010.11929). URL: <https://arxiv.org/abs/2010.11929>.
- [drC22] drCathieSo.eth. keras2circom. <https://github.com/socathie/keras2circom>. 2022.
- [Gra+21] Lorenzo Grassi et al. “Poseidon: A New Hash Function for Zero-Knowledge Proof Systems”. In: USENIX Security Symposium. 2021. URL: <https://api.semanticscholar.org/CorpusID:221069468>.
- [Gro16] Jens Groth. “On the Size of Pairing-Based Non-interactive Arguments”. In: Advances in Cryptology – EUROCRYPT 2016. Ed. by Marc Fischlin and Jean-Sebastien Coron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 305–326. ISBN: 978-3-662-49896-5.
- [GW20] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for zk-SNARKs. Cryptology ePrint Archive, Paper 2020/315. 2020. URL: <https://eprint.iacr.org/2020/315>.
- [Hab22] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530. 2022. URL: <https://eprint.iacr.org/2022/1530>.
- [He+15] Kaiming He et al. Deep Residual Learning for Image Recognition. 2015. arXiv: [1512.03385 \[cs.CV\]](https://arxiv.org/abs/1512.03385). URL: <https://arxiv.org/abs/1512.03385>.
- [Hou23] Youssef El Housni. “Pairings in Rank-1 Constraint Systems”. In: Applied Cryptography. Ed. by Mehdi Tibouchi and XiaoFeng Wang. Cham: Springer Nature Switzerland, 2023, pp. 339–362. ISBN: 978-3-031-33488-7.

- [How+19] Andrew Howard et al. Searching for MobileNetV3. 2019. arXiv: 1905.02244 [cs.CV]. URL: <https://arxiv.org/abs/1905.02244>.
- [HSS18] Jie Hu, Li Shen, and Gang Sun. “Squeeze-and-excitation networks”. In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2018, pp. 7132–7141.
- [ISY20] Takumi Ishiyama, Takuya Suzuki, and Hayato Yamana. Highly Accurate CNN Inference. 2020. arXiv: 2009.03727 [cs.LG]. URL: <https://arxiv.org/abs/2009.03727>.
- [Kan22] Daniel Kang. ddkang/zkml Framework. <https://github.com/ddkang/zkml>. 2022.
- [Lab25] Lagrange Labs. deep-prove. <https://github.com/Lagrange-Labs/deep-prove>. 2025.
- [Liu+24] Tianyi Liu et al. Ceno: Non-uniform, Segment and Parallel Zero-knowledge Virtual Machine. Cryptology ePrint Archive, Paper 2024/387. 2024. URL: <https://eprint.iacr.org/2024/387>.
- [Mor+19] Eduardo Morais et al. “A survey on zero knowledge range proofs and applications”. In: SN Applied Sciences 1.8 (2019), p. 946. DOI: 10.1007/s42452-019-0989-z. URL: <https://doi.org/10.1007/s42452-019-0989-z>.
- [OREHS24] Bjorn Oude Roelink, Mohammed El-Hajj, and Dipti Sarmah. “Systematic review: Comparing zk-SNARK, zk-STARK, and bulletproof protocols for privacy-preserving authentication”. In: SECURITY AND PRIVACY 7.5 (2024), e401. DOI: <https://doi.org/10.1002/spy2.401>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/spy2.401>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spy2.401>.
- [Rar25] Rarimo. Rarimo. <https://github.com/rarimo>. 2025.
- [Set19] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. Cryptology ePrint Archive, Paper 2019/550. 2019. URL: <https://eprint.iacr.org/2019/550>.
- [Sou+24] Tobin South et al. Verifiable evaluations of machine learning models using zkSNARKs. 2024. arXiv: 2402.02675 [cs.LG]. URL: <https://arxiv.org/abs/2402.02675>.
- [Wah+17] Riad S. Wahby et al. Doubly-efficient zkSNARKs without trusted setup. Cryptology ePrint Archive, Paper 2017/1132. 2017. URL: <https://eprint.iacr.org/2017/1132>.
- [Xie+25] Tiancheng Xie et al. zkPyTorch: A Hierarchical Optimized Compiler for Zero-Knowledge Machine Learning. Cryptology ePrint Archive, Paper 2025/535. 2025. URL: <https://eprint.iacr.org/2025/535>.
- [zca25] zcash. halo2. <https://zcash.github.io/halo2/>. 2025.

Appendices

A UltraGroth Security Proof

First, let us give the definitions of properties that the UltraGroth must satisfy in order to be a secure zk-SNARK protocol.

Definition. A zk-SNARK protocol $\Pi = (\text{Setup}, \text{Prove}, \text{Verify})$ may have the following properties:

1. Perfect Completeness. Π is complete if for any efficient relation \mathcal{R} the valid proof π is always accepted by the verifier:

$$\Pr \left[\text{Verify}(\text{vp}, \mathbb{x}, \pi) = 1 \mid \begin{array}{l} (\text{pp}, \text{vp}) \leftarrow \text{Setup}(1^\lambda, \mathcal{R}) \\ \pi \leftarrow \text{Prove}(\text{pp}, \mathbb{x}, \mathbb{w}) \\ (\mathbb{x}, \mathbb{w}) \in \mathcal{R} \end{array} \right] = 1$$

2. Perfect Zero-Knowledge. Π provides perfect zero-knowledge if for any efficient relation \mathcal{R} , there exists PPT simulator Sim such that for any $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, the distribution of a real proof is identical to the distribution of a simulated proof. Formally, after a trusted setup $(\text{pp}, \text{vp}) \leftarrow \text{Setup}(1^\lambda, \mathcal{R})$, the following two distributions are identical:

$$\{\pi \leftarrow \text{Prove}(\text{pp}, \mathbb{x}, \mathbb{w}) : \pi\} \approx_C \{\pi' \leftarrow \text{Sim}(\text{vp}, \mathbb{x}) : \pi'\}$$

3. Computational Knowledge Soundness. Π is a computational proof of knowledge if for any efficient relation \mathcal{R} and for any PPT malicious prover \mathcal{P}^* , there exists a PPT extractor \mathcal{E} which can “extract” the secret witness \mathbb{w} from \mathcal{P}^* . Formally, if a prover \mathcal{P}^* can produce a valid proof π for a statement \mathbb{x} with non-negligible probability, then the extractor (given oracle access to \mathcal{P}^*) can produce a valid witness \mathbb{w}^* for that statement:

$$\Pr \left[\begin{array}{l} \text{Verify}(\text{vp}, \mathbb{x}, \pi) = 1 \\ (\mathbb{x}, \mathbb{w}^*) \in \mathcal{R} \end{array} \mid \begin{array}{l} (\text{pp}, \text{vp}) \leftarrow \text{Setup}(1^\lambda, \mathcal{R}); \\ \pi \leftarrow \mathcal{P}^*(\text{pp}, \mathbb{x}); \\ \mathbb{w}^* \leftarrow \mathcal{E}^{\mathcal{P}^*}(\text{pp}, \mathbb{x}) \end{array} \right] \geq 1 - \text{negl}(\lambda)$$

Theorem. UltraGroth protocol is a secure zk-SNARK protocol, i.e., it satisfies perfect completeness, perfect zero-knowledge, and computational knowledge soundness.

Proof. Completeness is easy to show. Recall that the verifier has to check whether

$$e(\pi_A, \pi_B) = e(g_1^\alpha, g_2^\beta) \cdot e(\pi_{IC}, g_2^\gamma) \cdot \prod_{i \in [d+1]} e(\pi_C^{(i)}, g_2^{\delta_i})$$

Consider the expression $m_C := \prod_{i \in [d+1]} e(\pi_C^{(i)}, g_2^{\delta_i})$. Let $g_T := e(g_1, g_2)$, then:

$$\begin{aligned} m_C &= \prod_{i \in [d+1]} e(\pi_C^{(i)}, g_2^{\delta_i}) = g_T^{\sum_{i \in [d]} \delta_i c_i(\tau) + \delta_d c_d(\tau)} \\ &= g_T^{\sum_{i \in [d]} \delta_i \left(\sum_{j \in \mathcal{I}_W^{(i)}} z_j \frac{\zeta_j(\tau)}{\delta_i} + r_i \delta_d \right) + \delta_d c_d(\tau)} \\ &= g_T^{\sum_{i \in [d]} \sum_{j \in \mathcal{I}_W^{(i)}} z_j \zeta_j(\tau) + \delta_d \sum_{i \in [d]} \delta_i r_i + \delta_d c_d(\tau)} \\ &= g_T \end{aligned}$$

Now, notice that $\sum_{i \in [d]} \sum_{j \in \mathcal{I}_W^{(i)}}$ is essentially summing over index array $\bigcup_{i \in [d]} \mathcal{I}_W^{(i)}$, which in turn is the set $\mathcal{I}_W \setminus \mathcal{I}_W^{(d)}$. Additionally, we expand $c_d(\tau)$:

$$m_C = g_T^{\sum_{j \in \mathcal{I}_W \setminus \mathcal{I}_W^{(d)}} z_j \zeta_j(\tau) + \delta_d \sum_{i \in [d]} \delta_i r_i + \sum_{i \in \mathcal{I}_W^{(d)}} z_i \zeta_i(\tau) + h(\tau)t(\tau) + \delta_d a(\tau)s + \delta_d b(\tau)r - \delta_d \sum_{i \in [d]} r_i \delta_i - rs \delta_d^2}$$

Note that $\sum_{j \in \mathcal{I}_W \setminus \mathcal{I}_W^{(d)}} + \sum_{j \in \mathcal{I}_W^{(d)}} = \sum_{j \in \mathcal{I}_W}$. Hence, we further simplify the expression:

$$m_C = g_T^{\sum_{j \in \mathcal{I}_W} z_j \zeta_j(\tau) + h(\tau)t(\tau) + \delta_d a(\tau)s + \delta_d b(\tau)r - rs \delta_d^2}$$

But notice that this expression is nothing but

$$m_C = e\left(g_1^{\tilde{c}_d(\tau)}, g_2^{\delta_d}\right), \quad \tilde{c}_d(X) = \frac{1}{\delta_d} \left(\sum_{j \in \mathcal{I}_W} z_j \zeta_j(X) + h(X)t(X) \right) + a(X)s + b(X)r - rs \delta_d$$

This is exactly the expression that we had in the original Groth16's verifier equation, but instead of a single δ , only the last coefficient δ_d is used. Hence, the correctness of the UltraGroth verifier is guaranteed by the correctness of the original Groth16 verifier.

Now we show *perfect zero-knowledge*. Note that the pairing check verifies whether $a(\tau)b(\tau) = \alpha\beta + \text{IC}(\tau)\gamma + \sum_{i \in [d+1]} \delta_i c_i(\tau)$. For that reason, we propose the following simulator Sim that outputs a valid proof π that is indistinguishable from the real proof.

Simulator $\text{Sim}(\text{vp}, \mathbb{X}) \rightarrow \pi$:

1. Sample uniformly random $a', b', \{c'_i\}_{i \in [d]} \xleftarrow{R} \mathbb{F}$, corresponding to the exponents of corresponding proof elements.
2. Output the proof $\pi = (g_1^{a'}, g_2^{b'}, \{g_1^{c'_i}\}_{i \in [d+1]})$ where:

$$c'_d = \frac{a'b' - \alpha\beta - \sum_{j \in \mathcal{I}_X} z_j \zeta_j(\tau) - \sum_{i \in [d]} c'_i \delta_i}{\delta_d}$$

Let us show why it works. The proof correctness follows directly from the verification equation. Note that first $d+2$ components of the proof are uniformly distributed groups elements (since the corresponding powers are uniformly distributed) while the last term c'_d is computed uniquely from other coefficients; thus it is also uniformly random. Hence, the UltraGroth protocol is perfectly zero-knowledge.

Finally, we show *computational knowledge soundness*. We generalize the results closely following the original Groth16 paper [Gro16]. Recall that Groth16 is a pairing-based NIZK argument for QAP derived from NILP. UltraGroth, in turn, can also be viewed as an extension of NILP. That said, we simply have to show that any affine strategy that succeeds in non-negligible probability can be used to extract the witness. Recall that in the affine strategy, the prover computes the proof by applying the linear transformation on the common reference string: $\pi = \Pi\sigma = (a(\tau), b(\tau), \{c_i(\tau)\}_{i \in [d+1]}) \in \mathbb{F}^{d+3}$. Here,

$$\sigma = \left(\alpha, \beta, \gamma, \{\delta_i\}_{i \in [d+1]}, \{\tau^i\}_{i \in [n]}, \left\{ \frac{\zeta_i(\tau)}{\gamma} \right\}_{i \in \mathcal{I}_X}, \left\{ \frac{\zeta_j(\tau)}{\delta_i} \right\}_{i,j \in [d+1] \times \mathcal{I}_W^{(i)}}, \left\{ \frac{\tau^i t(\tau)}{\delta_d} \right\}_{i \in \mathcal{I}_W^{(d)}} \right)$$

Therefore, the prover's strategy for the first part a is following:

$$a = a_\alpha \alpha + a_\beta \beta + a_\gamma \gamma + \sum_{i \in [d+1]} a_\delta^{(i)} \delta_i + a(\tau) + \sum_{i \in \mathcal{I}_X} a_i \frac{\zeta_i(\tau)}{\gamma} + \sum_{i \in [d+1]} \sum_{j \in \mathcal{I}_W^{(d)}} a_j \frac{\zeta_j(\tau)}{\delta_i} + a_h(\tau) \frac{t(\tau)}{\delta_d},$$

where field elements $a_\alpha, a_\beta, a_\gamma, \{a_\delta^{(i)}\}_{i \in [d+1]}, \{a_i\}_{i \in \mathcal{I}_W}$ and polynomials $a(\tau)$ and $a_h(\tau)$ are known and correspond to the first row of the matrix Π . The same structure, albeit with different coefficients, holds for b and $\{c_i\}_{i \in [d+1]}$.

Now we view the verification equation as an equality of multivariate Laurent polynomials. By the Schwartz-Zippel lemma, the prover has a negligible probability for making the verification equation hold for randomly selected indeterminates $\alpha, \beta, \gamma, \{\delta_i\}_{i \in [d+1]}$ and τ . Now recall that the check is following:

$$ab = \alpha\beta + \text{IC}(\tau)\gamma + \sum_{i \in [d+1]} \delta_i c_i.$$

Now we are going to derive what are the coefficients of the corresponding polynomials a, b , and $\{c_i\}_{i \in [d+1]}$. The terms with indeterminate α^2 are $a_\alpha b_\alpha \alpha^2 = 0$ and so $a_\alpha = 0$ or $b_\alpha = 0$. Without loss of generality, assume $b_\alpha = 0$. Terms with $\alpha\beta$ gives us $a_\alpha b_\beta + a_\beta b_\alpha = a_\alpha b_\beta = 1$. Again, without loss of generality, assume $a_\alpha = 1$ and $b_\beta = 1$. Finally, the indeterminate β^2 gives $a_\beta b_\beta = a_\beta = 0$. Therefore, the expressions for a and b are now simplified down to:

$$a = \alpha + a_\gamma \gamma + \sum_{i \in [d+1]} a_\delta^{(i)} \delta_i + \dots, \quad b = \beta + b_\gamma \gamma + \sum_{i \in [d+1]} b_\delta^{(i)} \delta_i + \dots$$

Now consider the terms involving $1/\delta_d^2$. We have:

$$\left(\sum_{j \in \mathcal{I}_W^{(d)}} a_j \zeta_j(\tau) + a_h(\tau) t(\tau) \right) \left(\sum_{j \in \mathcal{I}_W^{(d)}} b_j \zeta_j(\tau) + b_h(\tau) t(\tau) \right) = 0$$

Without loss of generality, assume $\sum_{j \in \mathcal{I}_W^{(d)}} a_j \zeta_j(\tau) + a_h(\tau) t(\tau) = 0$. However, looking at term $\frac{\alpha}{\delta_d} \sum_{j \in \mathcal{I}_W^{(d)}} b_j \zeta_j(\tau) + a_h(\tau) t(\tau) = 0$ shows that in fact $\sum_{j \in \mathcal{I}_W^{(d)}} b_j \zeta_j(\tau) + a_h(\tau) t(\tau) = 0$ as well.

Now, consider the terms with $1/\delta_i^2$ for $i \neq d$. We have:

$$\sum_{j \in \mathcal{I}_W^{(i)}} a_j \zeta_j(\tau) \cdot \sum_{j \in \mathcal{I}_W^{(i)}} b_j \zeta_j(\tau) = 0$$

Without loss of generality assume $\sum_{j \in \mathcal{I}_W^{(d)}} a_j \zeta_j(\tau) = 0$. Again, looking at term $\frac{\alpha}{\delta_i} \sum_{j \in \mathcal{I}_W^{(i)}} b_j \zeta_j(\tau) = 0$ shows that in fact $\sum_{j \in \mathcal{I}_W^{(i)}} b_j \zeta_j(\tau) = 0$ as well.

Now terms involving $1/\gamma^2$ give us

$$\sum_{j \in \mathcal{I}_X} a_j \zeta_j(\tau) \cdot \sum_{j \in \mathcal{I}_X} b_j \zeta_j(\tau) = 0$$

For the same very reason, we derive that both sums are zero. Finally, the terms $a_\gamma \beta \gamma = 0$ and $b_\gamma \alpha \gamma = 0$ show that $a_\gamma = 0$ and $b_\gamma = 0$. Therefore, we have

$$a = \alpha + a(\tau) + \sum_{i \in [d+1]} a_\delta^{(i)} \delta_i, \quad b = \beta + b(\tau) + \sum_{i \in [d+1]} b_\delta^{(i)} \delta_i$$

Now, let us write the product ab explicitly:

$$\begin{aligned} ab &= \alpha\beta + \alpha b(\tau) + \alpha \sum_{i \in [d+1]} b_\delta^{(i)} \delta_i \\ &\quad + \beta a(\tau) + a(\tau)b(\tau) + a(\tau) \sum_{i \in [d+1]} b_\delta^{(i)} \delta_i \\ &\quad + \sum_{i \in [d+1]} a_\delta^{(i)} \delta_i \beta + \sum_{i \in [d+1]} a_\delta^{(i)} \delta_i b(\tau) + \sum_{i \in [d+1]} a_\delta^{(i)} \delta_i \sum_{j \in [d+1]} b_\delta^{(j)} \delta_j \\ &= \alpha\beta + \sum_{j \in \mathcal{I}_X} z_j (\beta \ell_i(\tau) + \alpha r_i(\tau) + o_i(\tau)) + \sum_{i \in [d+1]} \delta_i c_i \end{aligned}$$

Consider the terms that involve α (but not $\alpha \delta_i$ and such!) and β . We have:

$$b(\tau) = \sum_{j \in \mathcal{I}_X} z_j r_i(\tau) + \sum_{j \in \mathcal{I}_W} c_j r_j(\tau), \quad a(\tau) = \sum_{j \in \mathcal{I}_X} z_j \ell_i(\tau) + \sum_{j \in \mathcal{I}_W} c_j \ell_j(\tau)$$

Thus, define $z_j := c_j$ for $j \in \mathcal{I}_W$. This way, we have

$$a(\tau) = \sum_{j \in [n]} z_j \ell_j(\tau), \quad b(\tau) = \sum_{j \in [n]} z_j r_j(\tau)$$

Finally, looking at the terms involving powers of τ we have:

$$\sum_{j \in [n]} z_j \ell_j(\tau) \cdot \sum_{j \in [n]} z_j r_j(\tau) = \sum_{j \in [n]} z_j o_j(\tau) + c_h(\tau) t(\tau)$$

This finally shows that $\{z_j\}_{j \in \mathcal{I}_W} \equiv \{c_j\}_{j \in \mathcal{I}_W}$ is a witness for the statement $\mathbb{X} = \{z_j\}_{j \in \mathcal{I}_W}$.

B Quantization Scheme Proof

Theorem. Let $\hat{x} := Q_\rho(x)$ and $\hat{y} := Q_\rho(y)$ be the quantized values of x and y , respectively. Then, if we define circuit $C_f(\hat{x}, \hat{y}) = \hat{x} + \hat{y}$ for the addition $f(x, y) = x + y$, the following relation for an error ϵ_ρ holds:

$$\epsilon_\rho := |D_\rho(C_f(\hat{x}, \hat{y})) - f(x, y)| \leq 2 \cdot 2^{-\rho}$$

Consequently, the error ϵ_ρ is negligible in ρ .

Proof. For simplicity, assume $x, y > 0$ (the negative case, with more care, can be handled similarly). Since $\hat{x} = Q_\rho(x)$ and $\hat{y} = Q_\rho(y)$, we have $\hat{x} = 2^\rho x + \delta_x$ for $|\delta_x| < 1$. Same holds for y : we have $\hat{y} = 2^\rho y + \delta_y$ with $|\delta_y| < 1$. Then, it is easy to see that:

$$C_f(\hat{x}, \hat{y}) = \hat{x} + \hat{y} = (2^\rho x + \delta_x) + (2^\rho y + \delta_y) = 2^\rho(x + y) + (\delta_x + \delta_y)$$

Dequantization of such expression gives:

$$D_\rho(C_f(\hat{x}, \hat{y})) = x + y + 2^{-\rho}(\delta_x + \delta_y)$$

From which follows that $|D_\rho(C_f(\hat{x}, \hat{y})) - f(x, y)| \leq 2^{-\rho}(|\delta_x| + |\delta_y|) \leq 2 \cdot 2^{-\rho}$.

Theorem. Let $\hat{x} := Q_\rho(x)$ and $\hat{y} := Q_\rho(y)$ be the quantized values of x and y , respectively. Then, if we define circuit $C_f(\hat{x}, \hat{y}) = \hat{x} \cdot \hat{y}$ for the multiplication $f(x, y) = x \cdot y$ and assume $\beta := 2 \max\{|x|, |y|\}$ to be relatively small, the following relation for an error ϵ_ρ holds:

$$\epsilon_\rho := |D_{2\rho}(C_f(\hat{x}, \hat{y})) - f(x, y)| \leq 2^{-\rho}\beta + 2^{-2\rho}$$

Consequently, the error ϵ_ρ is negligible in ρ .

Proof. For simplicity, assume $x, y > 0$ (the negative case, with more care, can be handled similarly). Since β is assumed to be small, we assume that $\hat{x} = [2^\rho x]$ and $\hat{y} = 2^\rho y$ as well as $\hat{x} \cdot \hat{y}$ do not overflow the field. Then, it is easy to see that $\hat{x} = 2^\rho x + \delta_x$ for $|\delta_x| < 1$. Same holds for y : we have $\hat{y} = 2^\rho y + \delta_y$ with $|\delta_y| < 1$. Then,

$$C_f(\hat{x}, \hat{y}) = \hat{x} \cdot \hat{y} = (2^\rho x + \delta_x)(2^\rho y + \delta_y) = 2^{2\rho}xy + 2^\rho(\delta_x y + \delta_y x) + \delta_x \cdot \delta_y$$

Now, we can dequantize the result with precision 2ρ :

$$D_{2\rho}(C_f(\hat{x}, \hat{y})) = xy + 2^{-\rho}(\delta_x y + \delta_y x) + 2^{-2\rho}\delta_x \delta_y$$

Now note that $|\delta_x y + \delta_y x| \leq |x| + |y| \leq 2 \max\{|x|, |y|\} = \beta$. It is also easy to see that $|2^{-2\rho}\delta_x \delta_y| < 2^{-2\rho}$. Therefore,

$$\begin{aligned} \epsilon_\rho &= |D_{2\rho}(C_f(\hat{x}, \hat{y})) - f(x, y)| \\ &= |2^{-\rho}(\delta_x y + \delta_y x) + 2^{-2\rho}\delta_x \delta_y| \\ &\leq 2^{-\rho}|\delta_x y + \delta_y x| + 2^{-2\rho}|\delta_x \delta_y| \\ &\leq 2^{-\rho}\beta + 2^{-2\rho}. \blacksquare \end{aligned}$$