Exact Constrained-Training Neural Networks for Confidential 8-bit Arithmetic Primitives in Code Obfuscation

Ning Shi[a,d], Lei Xu[a], Tianqing Zhu[b], WanLei Zhou[b], Weizhi Meng[c], Yu-an Tan[a*]

[a]*School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, 100081, China*
[b]*Faculty of Data Science, City University of Macau, Macao Special Administrative Region of China, China*
[c]*School of Computing and Communications, Lancaster University, United Kingdom*
[d]*Hebei Key Laboratory of IoT Blockchain Integration, Shijiazhuang University, Shijiazhuang, 050035, China*

Abstract

Neural networks are rarely utilized for exact arithmetic computations.However, we aim to leverage them as accurate computing units in code obfuscation scenarios. The effectiveness of traditional code obfuscation methods is increasingly compromised by advanced reverse-engineering tools, which exploit the semantic transparency of arithmetic operations. To address this, we propose constrained-training neural networks tailored to 8-bit integer addition and multiplication, the most common operations in security-critical software. For addition, we introduce a constrained training algorithm that integrates weight clipping, linearity-enforcing loss terms, and boundary-case oversampling, enabling convergence to 100% accuracy across the entire domain. For multiplication, we design the adaptive symbol-gated NALU (ASG-NALU), an improved 4-bit multiplier that achieves exact results with reduced complexity. Combined with a cascade decomposition strategy, it extends to 8-bit multiplication with guaranteed correctness. Experiments confirm 100% in-domain accuracy, while out-of-domain inputs trigger catastrophic failures that act as natural traps, providing hidden security checks against dynamic analysis. These results establish exact constrained-training neural networks as confidential arithmetic primitives and firmly position neural arithmetic as a promising approach for advancing code obfuscation techniques.

Keywords: Code obfuscation, Neural arithmetic, Cascade decomposition, Domain opacity, Confidential arithmetic primitives

1. Introduction

Code obfuscation is a fundamental technique in software protection, aiming to prevent reverse engineering, safeguard intellectual property, and mitigate attacks against embedded and security-critical systems [1, 2]. Traditional methods include control-flow obfuscation, mixed Boolean-arithmetic (MBA) transformations, and virtualization-based protections. While effective in earlier decades, these approaches are increasingly undermined by advances in program analysis, symbolic execution, and modern reverse engineering frameworks such as IDA, Ghidra, and large language model (LLM)-based decompilers [3, 4]. Recent studies have shown that MBA transformations, for example, can be systematically dismantled using algebraic simplification or search-based methods such as Monte Carlo tree search (MCTS) [5, 6, 7]. As a result, new directions are urgently needed to construct obfuscation techniques that resist both classical and machine-learning-assisted attacks.

In parallel, neural networks have demonstrated their ability to approximate symbolic functions, including arithmetic operators, through specialized architectures such as the Neural Accumulator (NAC) and Neural Arithmetic Logic Unit (NALU) [8, 9]. However, these models were primarily designed for extrapolation in algorithmic learning tasks and often sacrifice precision within bounded integer domains [10, 11]. For obfuscation purposes, however, precision is paramount: a single incorrect output may compromise program correctness or reveal implementation flaws exploitable by attackers. Thus, the challenge is not to generalize indefinitely, but to ensure *exact correctness within a specified integer domain*, while simultaneously leveraging the natural limitations of neural networks as a security feature.

In this paper, we present a novel approach to code obfuscation based on accurate neural computing units. Instead of encoding arithmetic operations through syntactic rewriting or virtualization, we replace integer addition and multiplication with neural networks that achieve perfect correctness within predefined domains. This strategy introduces an additional semantic layer of obfuscation: attackers face the dual challenge of identifying the neural operators and reconstructing their exact functional scope. Furthermore, the restricted generalization of neural models, often seen as a drawback in machine learning, is here reinterpreted as a *security barrier*.

Our contributions are summarized as follows:

- **Exact neural addition under constrained training.** We propose a constrained training algorithm for 8-bit addition that combines hybrid loss, adaptive weight clipping, and boundary-case sampling, achieving 100% accuracy across the full input domain.
- **Confidential multiplication via cascade decomposition.** We introduce a cascade strategy that reduces 8-bit multiplication to 4-bit subproblems and propose the Adaptive Symbol-Gated NALU (ASG-NALU), which attains 100% accuracy with far fewer parameters and iterations than vanilla NALU, while concealing intermediate computations to enhance confidentiality.
- **Unified evaluation for confidential primitives.** We define three key metrics for neural arithmetic units: *correctness* (in-domain accuracy), *security* (trap failure rates), and *practicality* (runtime efficiency), providing a basis for benchmarking obfuscation-oriented models.
- **Extrapolation failure as defense.** We are the first to systematically reinterpret the well-known failure of neural networks beyond training ranges as a *security barrier*.Hidden training ranges create natural traps that mislead fuzzing, symbolic execution, and synthesis, while training over disjoint ranges further fragments the domain and complicates reverse engineering.

2. Related Work

2.1. Traditional Code Obfuscation Techniques

Code obfuscation has long been adopted as a defense mechanism against reverse engineering and software tampering. Classic approaches can be broadly categorized into three groups: *control-flow obfuscation*, *data obfuscation*, and *virtualization-based protection*. Control-flow obfuscation modifies program structure by techniques such as flattening, opaque predicates, and loop transformations [12, 13], making the control graph less interpretable to human analysts and automated tools. Data obfuscation strategies include constant hiding and mixed Boolean-arithmetic (MBA) transformations, which encode simple arithmetic operations into convoluted logical-arithmetic forms [14, 15]. Virtualization-based obfuscation replaces native instructions with custom bytecode executed by an embedded virtual machine [16, 17, 18], creating a semantic gap that complicates decompilation. Additionally, encryption-based methods such as white-box cryptography [19] and trusted execution environments [20, 21, 22] have been deployed to protect sensitive algorithms.

Despite their widespread industrial use, these methods are increasingly vulnerable. Advances in reverse-engineering tools such as IDA Pro [23], Ghidra [3], and more recently LLM-based decompilers [24, 25], have significantly lowered the barrier for automated de-obfuscation. Static pattern matching, symbolic execution, and AI-assisted reasoning can efficiently recover program semantics from even heavily obfuscated binaries [26, 27, 28]. As a result, traditional strategies relying on syntactic distortion or VM-level indirection are facing diminishing security returns, motivating exploration into computation-level obfuscation.

2.2. Mixed Boolean-Arithmetic Transformations

MBA transformations represent one of the most widely deployed obfuscation mechanisms for arithmetic expressions. By encoding integer addition and multiplication with mixed logical and arithmetic operators [29], MBA seeks to increase algebraic complexity and defeat signature-based detection. MBA-based encodings significantly raise the difficulty of static simplification and symbolic reasoning, and for a period were considered effective against standard reverse-engineering pipelines.

However, subsequent research has revealed systematic weaknesses in MBA encodings. Monte Carlo tree search (MCTS) [30, 31], algebraic reduction methods [29], and SAT-based solvers [32] have proven effective in recovering canonical arithmetic expressions. Moreover, MBA suffers from boundedness: in small integer domains (e.g., 8-bit or 32-bit arithmetic), exhaustive enumeration makes MBA particularly fragile against automated de-obfuscation [5, 33]. Thus, while MBA provides lightweight obfuscation with low runtime overhead, it lacks resilience against advanced analysis tools.

2.3. Machine Learning for Code Obfuscation

To overcome the limitations of handcrafted obfuscation, machine learning (ML) techniques have been introduced. Reinforcement learning has been employed to adaptively select obfuscation strategies depending on the target program and attacker model [34, 35]. Generative models, including generative adversarial networks (GANs), have been leveraged to mutate code and produce diverse program variants for malware evasion [36, 37]. In malware protection specifically, LLVM-based obfuscation combined with ML classifiers has been studied to resist signature detection, with function cloning and code morphing shown to evade conventional antivirus pipelines [38, 39]. More recently, approaches such as CodeCipher propose ML-driven source code transformations to evade detection by large language models [40].

These methods demonstrate that ML can automate structural diversity and increase adversarial robustness. However, most efforts operate on high-level code structure (e.g., control flow, abstract syntax trees) rather than ensuring exactness of arithmetic computations. This leaves a gap in the design of computation-level obfuscation strategies.

2.4. Neural Arithmetic Units

Neural arithmetic computation has been actively studied in the ML community through architectures such as the Neural Accumulator (NAC) and the Neural Arithmetic Logic Unit (NALU) [8], as well as extensions like the Neural Arithmetic Unit (NAU) [41],and Neural power units(NPU) [42]. These models aim to learn operations such as addition, subtraction, multiplication, and division in a differentiable framework, and have found applications in algorithmic learning, symbolic regression, and extrapolation tasks.

Nevertheless, their precision is limited in bounded integer domains. Rounding errors, unstable extrapolation, and failure to guarantee exact correctness make them unsuitable for contexts requiring bit-exact computations [43, 10, 11]. More importantly, these neural arithmetic units have not been considered in code protection. Whereas existing studies emphasize generalization, obfuscation scenarios require *exactness within a fixed integer range* (e.g., 8-bit signed arithmetic) rather than open-ended extrapolation. This mismatch motivates the adaptation and redesign of neural arithmetic models for obfuscation.

2.5. Neural Networks for Obfuscation and Security

The idea of leveraging neural networks themselves as obfuscation primitives is relatively novel. Early works examined adversarial perturbations to mislead malware classifiers or de-obfuscation tools [44, 45]. Some approaches suggested embedding neural networks as opaque functions to hinder symbolic reasoning and static analysis [46, 47]. Yet, these attempts are largely exploratory, and none directly address the possibility of implementing core arithmetic operations—such as integer addition and multiplication—via neural networks.

Replacing explicit arithmetic operators with neural inference would introduce a fundamentally new obfuscation layer. Attackers would face both the black-box complexity of neural models and the bounded-domain correctness constraints, complicating reverse engineering and algebraic simplification. To the best of our knowledge, no prior work has demonstrated such a computation-oriented obfuscation paradigm.

In summary, prior research spans three main directions: (i) traditional syntactic and structural obfuscation (control flow, MBA, virtualization), (ii) ML-driven structural transformations for diversity and evasion, and (iii) neural arithmetic units designed for generalization in algorithmic learning. None of these directly address the need for exact neural arithmetic computation tailored to bounded integer domains in obfuscation contexts. Our work fills this gap by introducing neural networks capable of performing addition and multiplication with perfect accuracy in 8-bit ranges, thereby enabling a novel form of computation-level code obfuscation.

3. Methodology for Confidential Arithmetic Primitives

3.1. Neural Addition Units

Addition of 8-bit integers $(a+b)$ produces results in the range $[-256, 254]$, where $a, b \in [-128, 127]$. Ensuring exact computation is critical for obfuscation scenarios, as even a single bit error can completely break the functionality of the obfuscated system. To address this, we developed a constrained training framework that enforces numerical precision across all possible input pairs.

3.1.1. Network Architectures

We evaluate two distinct neural network architectures engineered for the obfuscation of integer addition, specifically designed to achieve an optimal balance among precision, computational efficiency, and resistance to reverse engineering.As shown in Figure 1 and 2,we use common feedforward neural networks(FFNN-Add) as the benchmark and Neural Accumulators(NAC-Add) as the comparison architecture. The input is limited to two eight bit integers, and the intermediate layers of each network include n hidden nodes.

The output is the sum of two numbers.In the following chapters, we will provide a detailed introduction to two types of network structures and their respective details.

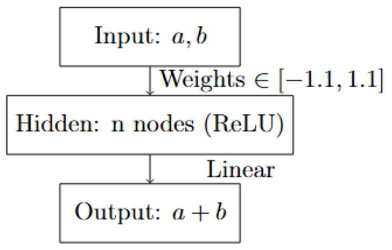

Figure 1: FFNN-Add Architecture

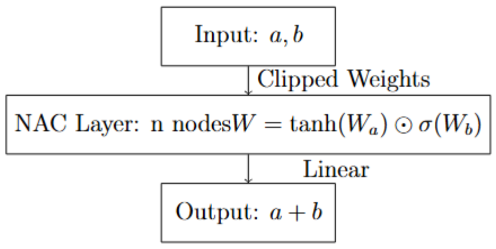

Figure 2: NAC-Add Architecture

3.1.2. FFNN-Add: Feedforward Neural Network for Addition

The **FFNN-Add** architecture is a three-layer feedforward neural network (**FFNN**) meticulously designed for computational efficiency.

- **Input Layer:** Comprises two nodes, directly accepting 8-bit integers a and b as inputs.
- **Hidden Layer:** Consists of n nodes employing the Rectified Linear Unit (**ReLU**) activation function. The strategic introduction of this non-linearity is crucial for preventing trivial reversal of the obfuscation through linear decomposition.
- **Output Layer:** Features a single node with a linear activation function. This choice ensures the preservation of integer values without scaling, maintaining an output range consistent with $[-256, 254]$.

3.1.3. NAC-Add: Neural Accumulator for Addition

The **NAC-Add** architecture is based on Neural Accumulators (**NAC**) [8], an approach inherently suited for arithmetic operations due to its structural obfuscation properties.

- **Input Layer:** Identical to FFNN-Add, with two nodes receiving 8-bit integers a and b.

- **NAC Layer:** Composed of n nodes. A distinctive weight constraint, $W = \tanh(W_a) \odot \sigma(W_b)$, where $\odot$ denotes element-wise multiplication, is applied. This constraint forces all weights to lie within the range $[-1, 1]$, thereby facilitating additive behavior while significantly obscuring direct interpretability of individual weights.

- **Output Layer:** A single node with a linear activation function, guaranteeing that the final output is an unadulterated sum of the activations from the preceding NAC layer.

NAC-Add is preferred for obfuscation tasks due to its non-interpretable weight structure, which makes reverse engineering significantly harder. In contrast, FFNN-Add serves as a baseline for efficiency, with fewer parameters and faster inference times, though its simpler weight distribution offers less inherent resistance to deobfuscation. Both architectures are trained using Adam optimizer with a learning rate of 10^{-4} and batch size of 2048, With training terminating only when training accuracy reaches 100% across all test inputs.

3.1.4. Constrained Training Algorithm for Guaranteed Accuracy and Convergence

To ensure a neural network achieves perfect, 100% accuracy for integer addition—a critical requirement for obfuscation—we must overcome the inherent imprecision of standard neural network training. Our approach utilizes a constrained training algorithm built upon three core mechanisms that enforce exactness, linearity, and robustness across all possible inputs.

1. **Hybrid Loss Function** We employ a hybrid loss function, $\mathcal{L}$, strategically designed to balance the competing demands of exactness and linearity, which is fundamental to accurate arithmetic. The function is defined as:

$$\mathcal{L} = \underbrace{\text{MAE}(y_{\text{pred}}, y_{\text{true}})}_{\text{Exactness Term}} + \underbrace{\lambda \|W\|_1}_{\text{Linearity Term}}$$

   $\text{MAE}(y_{\text{pred}}, y_{\text{true}})$, is the primary driver for correctness, forcing the network to minimize the difference between its predictions and the ground truth. This ensures the output is an exact integer value. Simultaneously, the linearity term, an L_1 penalty ($\lambda \|W\|_1$ with $\lambda = 0.01$) on the network's weights, W, discourages large weight values. This regularization promotes a more linear and interpretable network behavior, which is essential for faithfully performing a linear operation like addition.

2. **Adaptive Weight Clipping** A pivotal mechanism for maintaining perfect accuracy is adaptive weight clipping. This technique acts as a safeguard against the network's weights drifting into configurations that introduce non-linear mappings and approximation errors. After each gradient update, all weights are dynamically clipped to a narrow range of $[-1.1, 1.1]$. The specific threshold of 1.1 is a calculated choice: it is tight enough to strictly enforce linearity but wide enough to allow for the small weight adjustments necessary for the network to learn. This mechanism guarantees the network's ability to perform flawless, bit-perfect arithmetic computations across the entire input domain.
3. **Exhaustive Boundary Sampling** Ensuring 100% accuracy across all 2^{16} possible input pairs requires more than just a large dataset; it demands a robust strategy to handle boundary cases where errors are most likely to occur. Our training dataset includes all $256 \times 256 = 65,536$ possible 8-bit integer pairs. Furthermore, we employ a deliberate **oversampling of boundary cases**, such as $127 + 127$ (maximum positive sum) and $-128 + 127$ (a challenging overflow scenario). This exhaustive and biased sampling strategy is crucial for bolstering the network's stability and eliminating potential vulnerabilities. By guaranteeing stability and precision for even the most challenging or infrequent inputs, this approach is indispensable for achieving perfect accuracy in obfuscation-critical applications.

The process of training the neural addition unit algorithm is shown in Algorithm 1.

3.2. 4-bit Multiplication Architecture

We adopt the neural arithmetic logic unit (NALU) [41] as the foundational model for neural network-based multiplication. Although the original NALU-Mul demonstrates strong performance in 4-bit multiplication tasks, it exhibits significant limitations when handling negative numbers, severely impacting performance in signed arithmetic scenarios. Specifically, NALU-Mul often fails to maintain result accuracy with negative operands, producing outputs that deviate substantially from the ground truth. This deficiency arises mainly from its inadequate sign-processing mechanism: encoding and

Algorithm 1 Constrained Training Algorithm

```
Require: Training dataset $\mathcal{D} = \{(a_k, b_k, y_{\text{true},k})\}$, Learning rate $\alpha$, Regularization parameter $\lambda = 0.01$, Clipping threshold $T = 1.1$, Number of epochs $N_{\text{epochs}}$
Ensure: Trained Add network weights $W$ and biases $B$
 1: Initialize network weights $W$ and biases $B$ using He initialization for hidden layers and small random values for output layer.
 2: for epoch = 1 to $N_{\text{epochs}}$ do
 3:     Shuffle training dataset $\mathcal{D}$.
 4:     for each mini-batch $(a, b, y_{\text{true}})$ in $\mathcal{D}$ (with edge-case oversampling) do
 5:         Compute predicted output $y_{\text{pred}}$ using current $W, B$.
 6:         Calculate Exactness Term: $L_{\text{MAE}} = \text{MAE}(y_{\text{pred}}, y_{\text{true}})$.
 7:         Calculate Linearity Term: $L_{L1} = \lambda \|W\|_1$.
 8:         Calculate Hybrid Loss: $\mathcal{L} = L_{\text{MAE}} + L_{L1}$.
 9:         Compute gradients $\nabla_W \mathcal{L}$ and $\nabla_B \mathcal{L}$ via backpropagation.
10:         Update weights and biases:
11:         $W \leftarrow W - \alpha \nabla_W \mathcal{L}$
12:         $B \leftarrow B - \alpha \nabla_B \mathcal{L}$
13:                                               ▷ Adaptive Weight Clipping
14:         Clip weights: $W_{ij} \leftarrow \max(-T, \min(T, W_{ij})) \quad \forall W_{ij} \in W$.
15:     end for
16: end for
```

manipulating sign information for negative values is error-prone, especially in cases involving multiple negative operands. Furthermore, the model struggles to independently process sign bits apart from magnitude information, leading to misclassification of results' polarity in complex edge cases.

To address these challenges, we propose the **Adaptive Symbol-Gated NALU** (ASG-NALU), which preserves the core arithmetic flexibility of NALU while introducing three key enhancements: (i) symbol-aware gating, (ii) hierarchical attention over input pairs, and (iii) multiplication-constrained training. These innovations improve signed arithmetic robustness, optimize computational efficiency, and enhance obfuscation for secure applications.

The ASG-NALU architecture (Figure 3) builds on NALU-Mul while refining its structure for multiplicative accuracy and signed arithmetic robustness:

- **Input Layer:** Two input nodes represent 4-bit signed integers $x, y \in$

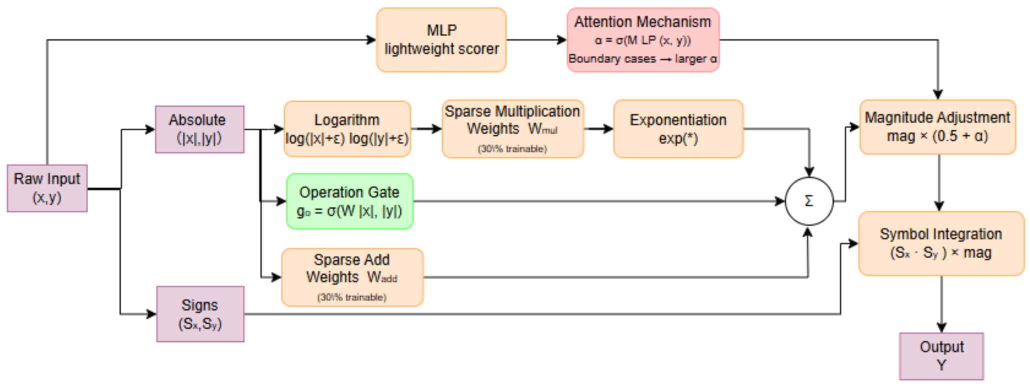

Figure 3: Adaptive Symbol-Gated NALU Multiplication Architecture

$[-8, 7]$, augmented with a *symbol preprocessing module* that extracts the signs $s_x = \text{sign}(x) \in \{-1, 1\}$ and $s_y = \text{sign}(y) \in \{-1, 1\}$, and absolute values $|x|, |y| \in [0, 8]$. This decouples sign handling from magnitude computation, mitigating the sign errors in vanilla NALU caused by log-transforms discarding sign information (e.g., -3×-4).

- **Hidden Layers:** A two-layer design with *adaptive node scaling* (64–128 nodes) to remove redundancy. *Sparse multiplicative connections* (30% trainable weights) enforce efficiency. The first hidden layer (*magnitude NALUs*) processes $|x|, |y|$ via multiplicative pathways, while the second layer (*symbol integrators*) fuses magnitudes with precomputed signs.

- **Output Layer:** A single output node produces $x \times y$, with a *clamping mechanism* enforcing integer outputs in $[-64, 63]$, preventing floating-point drift in downstream 8-bit recombination.

3.2.1. Adaptive Symbol-Gated Activation

The core innovation is the *dual-gate mechanism*, splitting the traditional NALU gate g into a *symbol gate* (g_s) and an *operation gate* (g_o) for robust signed arithmetic:

$$\text{ASG-NALU}(x,y) = \underbrace{(s_x \odot s_y)}_{\text{Sign Fusion}} \odot \Big[g_o \odot \exp\big(W \log(|x|+\varepsilon) + W' \log(|y|+\varepsilon)\big) + (1-g_o) \odot \big(V|x| + V'|y|\big) \Big] \tag{1}$$

where:

- $s_x \odot s_y$ explicitly computes the output sign via XOR-like logic (positive if $s_x = s_y$, negative otherwise), eliminating sign errors from the original NALU.
- $g_o \in [0,1]$ controls the multiplicative ($g_o \approx 1$) vs. additive ($g_o \approx 0$) pathways, initialized via *range-aware biasing* ($g_o \approx 0.8$ for $|x|, |y| \geq 1$; $g_o \approx 0.2$ for zeros).
- W, W', V, V' are sparsified (30% density) via L_0 regularization to reduce overfitting and enhance obfuscation.

3.2.2. Cascade Attention for Boundary-Case Robustness

To handle edge cases (e.g., $0 \times y$, -8×7), a *case-aware attention module* dynamically weights hidden layer activations: A lightweight attention scorer (2-layer MLP) computes $\alpha(x,y) \in [0,1]$, with $\alpha \approx 1$ for edge cases and $\alpha \approx 0.5$ for typical cases.The coefficient α scales activations in the second hidden layer, amplifying focus on challenging inputs without increasing model capacity.

3.2.3. Multiplication-Constrained Training

We replace the standard MAE loss with a **hybrid multiplication loss** to enforce exact 4-bit multiplication:

$$\mathcal{L} = \text{MAE}(y_{\text{pred}}, y_{\text{true}}) + \lambda_1 \|y_{\text{pred}} - x \times y\|_2^2 + \lambda_2 \|g_o - \mathbb{I}(|x|, |y|)\|_1 \tag{2}$$

where:

- The quadratic term ($\lambda_1 = 0.1$) heavily penalizes multiplication errors.
- $\mathbb{I}(|x|, |y|)$ is an indicator: 1 if $|x|, |y| \geq 1$, 0.3 if $x = 0$ or $y = 0$.
- $\lambda_2 = 0.05$ aligns g_o with expected operation modes, effectively acting as a "multiplication detector".

3.3. Cascade Decomposition Multiplication

The multiplication of two 8-bit signed integers ($a \times b$) presents unique challenges for neural network-based learning, primarily due to the vast range of possible outcomes. For 8-bit signed integers,where $a, b \in [-128, 127]$,the product $a \times b$ spans $[-16256, 16384]$, encompassing 32641 distinct values. This large output space, combined with the 65536 possible input pairs (a, b), creates a complex mapping that is intractable for direct end-to-end training. Neural networks trained on such a space struggle to achieve 100% accuracy due to the sparsity of edge-case examples (e.g., -128×127) and the non-linearities required to model extreme values.

To tackle this challenge, we put forward a cascade decomposition strategy that dissects the 8-bit multiplication problem into smaller, more manageable subproblems. This decomposition not only lowers computational complexity but also boosts obfuscation efficiency: by concealing the final product within a series of intermediate subcomputations, reverse-engineering the overall multiplication logic becomes substantially more difficult. Importantly, the decomposition allows for exhaustive training of subcomponents, guaranteeing precise results across all possible inputs—a prerequisite for reliable arithmetic operations.

3.3.1. Cascade Strategy

The core of our approach lies in splitting each 8-bit integer into two 4-bit signed components: a high-order half and a low-order half. Formally, for any 8-bit integers a and b, we decompose them as:

$$a = a_H \times 16 + a_L, \quad b = b_H \times 16 + b_L$$

where a_H (high-order half of a) and a_L (low-order half of a) are 4-bit signed integers, and similarly for b_H and b_L. This decomposition leverages the positional notation of binary integers: the factor of 16 arises because 4 bits correspond to $2^4 = 16$ possible values. For 4-bit signed integers, the range is $[-8, 7]$, ensuring symmetry around zero and avoiding overflow in subcomputations.

Expanding the product $a \times b$ using the decomposed forms reveals how to recombine the subproducts as Figure 4:

$$\begin{aligned} a \times b &= (a_H \times 16 + a_L)(b_H \times 16 + b_L) \\ &= a_H \times b_H \times 16^2 + (a_H \times b_L + a_L \times b_H) \times 16 + a_L \times b_L. \end{aligned}$$

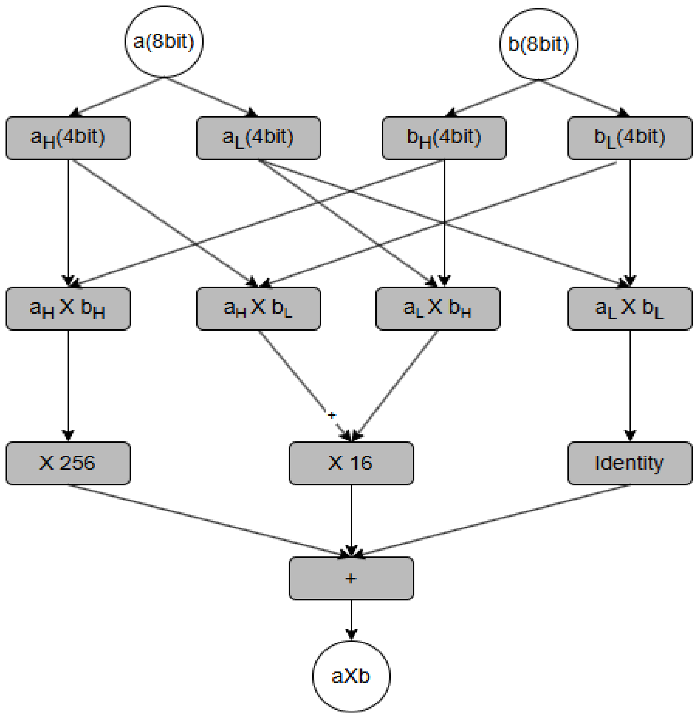

Figure 4: Decomposition for 8-bit Multiplication

Simplifying the powers of 16, this reduces to the final recombination formula:

$$a \times b = (a_H \times b_H) \times 256 + (a_H \times b_L + a_L \times b_H) \times 16 + (a_L \times b_L).$$

This decomposition transforms the 8-bit multiplication problem into four 4-bit multiplication subproblems: $a_H \times b_H$, $a_H \times b_L$, $a_L \times b_H$, and $a_L \times b_L$. Each subproblem involves 4-bit inputs from $[-8, 7]$, resulting in only $16 \times 16 = 256$ unique input pairs per subproblem—small enough to be exhaustively trained. The intermediate results of these subproblems are then scaled by powers of 16 (256 for the high-order subproduct, 16 for the low-order subproduct) and summed to yield the final 8-bit product. This hierarchical approach ensures that each component is learned exactly, with no information loss during recombination.

4. Experimental Evaluation

4.1. Experimental Setup

To rigorously evaluate both functional correctness and obfuscation robustness, two complementary data subsets were constructed, as summarized in Table 1.

Data Subset	**Description**
In-domain (ID)	Complete Cartesian products of 8-bit integer pairs for addition (65,536 pairs) and 4-bit integer pairs for multiplication (256 pairs), covering the entire valid operand space.
Trap Dataset	Designed for robustness testing beyond the training domain: *T1* — 10,000 random pairs from $[-500, 500]$; *T2* — 10,000 random pairs from $[-2000, 2000]$; *T3* — 10,000 random pairs from $[-5000, 5000]$; *T4* — 10,000 random pairs from $[-10{,}000, 10{,}000]$.

Table 1: Data subsets for functional correctness and obfuscation robustness evaluation

The *In-domain* subset contains the complete Cartesian products of all possible operand pairs within the target bitwidths: $2^8 \times 2^8 = 65{,}536$ pairs for addition and $2^4 \times 2^4 = 256$ pairs for multiplication. This exhaustive coverage ensures that the models are trained on the full spectrum of valid arithmetic cases. The *Trap* subset is designed to probe generalization limits and resilience under adversarial conditions. It comprises four distinct distributions: *T1* — 10,000 random pairs from $[-500, 500]$; *T2* — 10,000 random pairs from $[-2000, 2000]$; *T3* — 10,000 random pairs from $[-5000, 5000]$; *T4* — 10,000 random pairs from $[-10{,}000, 10{,}000]$, which significantly exceeds the target bitwidths to simulate aggressive black-box and fuzz testing.

All experiments were conducted on a workstation equipped with an Intel Core i7-12700 CPU (2.1 GHz), 32 GB RAM, and running Windows 11. Model training and evaluation were implemented in Python 3.10 with PyTorch 2.0.1.

4.2. Model Selection

To achieve robust arithmetic obfuscation, model architectures were carefully selected according to three core criteria: *computational accuracy* (to preserve functional correctness), *structural opacity* (to resist reverse engineering), and *efficiency* (to minimize runtime overhead). Accuracy ensures that the obfuscated program remains exact and reliable, opacity guarantees that the hidden arithmetic logic cannot be easily recovered by symbolic execution or algebraic simplification, and efficiency determines whether the approach is practical for integration into real-world systems. Balancing these three dimensions requires exploring both lightweight feed-forward baselines and structurally complex neural arithmetic units. Table 2 details the chosen architectures and their design rationales.

Table 2: Model Architectures and Select Rationale

Model	**Layer Nodes**	**Design Rationale**
FFNN-Add-1	2-8-1	Compact feed-forward model for 8-bit addition, reaching 100% accuracy. Serves as baseline.
FFNN-Add-2	2-16-1	Larger hidden layer doubles parameters, improving accuracy stability and raising reverse difficulty.
NAC-Add-1	2-8-1	Employs Neural Accumulator units with gated weights, adding structural opacity beyond standard FFNN.
NAC-Add-2	2-16-1	Expanded NAC with higher capacity. Enlarged weight space increases ambiguity and strengthens obfuscation.
NALU-Mul-1	2-64-64-1	NALU tailored for multiplication. Two 64-node layers ensure convergence to exact results with robustness.
NALU-Mul-2	2-128-128-1	Higher-capacity NALU variant doubling nodes, enhancing weight entropy and resistance to symbolic analysis.
ASGNALU-Mul-1	2-64-1	Adaptive Symbol-Gated NALU for multiplication. Gating improves opacity while preserving 100% accuracy.
ASGNALU-Mul-2	2-128-1	Scaled ASG-NALU with 128 nodes, offering improved stability and greater resistance to extraction attacks.

Model size directly impacts both obfuscation strength and computational overhead. Smaller models (e.g., FFNN-Add-1, NALU-Mul-1) with fewer parameters are ideal for performance-critical modules (e.g., real-time sensor processing).Conversely,larger models (e.g., NAC-Add-2,NALU-Mul-2) exhibit higher resistance to reverse engineering due to high-dimensional weight spaces and greater entropy, but incur longer inference times. These models are better suited for security-critical components such as license verification and cryptographic key generation, where functional secrecy outweighs latency constraints.

4.3. Experiment Result

4.3.1. Overview of model training

Table 3 demonstrates that, under our training regimen, all models attain 100% in-domain accuracy for algorithmic addition and multiplication, while also revealing how architectural bias and objective design shape efficiency.

Table 3: Model Training Data In Domain Range

Model	Parameters Count	Iterations	Weight Entropy
FFNN-Add-1	33	24	4.98 bits
FFNN-Add-2	65	90	4.65 bits
NAC-Add-1	40	38	5.05 bits
NAC-Add-2	80	24	5.82 bits
NALU-Mul-1	21440	1700	6.1517 bits
NALU-Mul-2	83840	1495	6.3398 bits
ASGNALU-Mul-1	802	95	5.7668 bits
ASGNALU-Mul-2	1570	154	4.9260 bits
Native arithmetic: Yes (no range restriction), no iterative training required			

* Train configuration:Adam optimize;batch size = 2048; constrained addition learning;random seed = 42.

For addition, plain feed-forward networks (FFNN-Add-1/2; 33/65 parameters) require 24/90 iterations, whereas NAC-Add-1/2 (40/80 parameters) converge in 38/24 iterations, showing that NAC's arithmetic inductive bias substantially accelerates convergence with only marginal parameter overhead. For multiplication, vanilla NALU also attains 100% accuracy but at a steep cost: NALU-Mul-1 (21,440 parameters) requires 1,700 iterations, and NALU-Mul-2 (83,840 parameters) requires 1,495 iterations. By contrast, ASGNALU-Mul achieves the same accuracy with pronounced efficiency gains—ASGNALU-Mul-1 uses just 802 parameters and 95 iterations ($\approx 26.7\times$ fewer parameters and $\approx 17.9\times$ fewer iterations vs. NALU-Mul-1), while ASGNALU-Mul-2 uses 1,570 parameters and 154 iterations ($\approx 53.4\times$ and $\approx 9.7\times$ reductions vs. NALU-Mul-2). Weight-entropy statistics corroborate these trends: FFNN/NAC entropies remain relatively low (4.98–6.14 bits), NALU's are consistently higher (6.1517–6.3398 bits), and ASGNALU remains moderate or lower (5.7668 and 4.9260 bits), consistent with sparse masks, attention, and gating mechanisms that concentrate effective degrees of freedom; the lower entropy reflects parameter efficiency rather than underfitting, given perfect accuracy across all models.

Overall, for addition, compact FFNNs are sufficient to guarantee correctness, whereas NAC-Adds, under comparable parameter counts, achieve higher weight entropy and thus greater arithmetic complexity. For multiplication, ASGNALU provides frontier-leading parameter–iteration efficiency, matching accuracy while using far fewer resources. While native arithmetic is trivially perfect without training, the value of these components lies in their differentiability and seamless integration into larger neural systems.

4.3.2. Effect of constrained training algorithm for addition

To validate the effectiveness of our constrained training algorithm, we conducted a comparative study between unconstrained and constrained training settings using the same FFNN-Add and NAC-Add architecture. Under the unconstrained setting, the network struggled to converge to perfect accuracy: even after hundred training epochs, the model still exhibited minor but non-negligible errors, especially on certain boundary cases. This observation confirms that standard gradient-based optimization alone is insufficient for guaranteeing exact integer arithmetic.

As shown in Figures 5 and 6, when trained with our proposed constrained algorithm—incorporating the hybrid loss function, adaptive weight clipping, and exhaustive boundary sampling—the same networks consistently reached 100% accuracy much earlier and with greater stability. For FFNN-Add-1, unconstrained learning required 111 epochs to achieve perfect accuracy, whereas the constrained variant converged in only 24 epochs. Similarly, FFNN-Add-2 required 174 epochs without constraints but only 90 epochs under the constrained setting. The effect is even more pronounced for NAC models: NAC-Add-1 converged at epoch 38 with constraints compared to 139

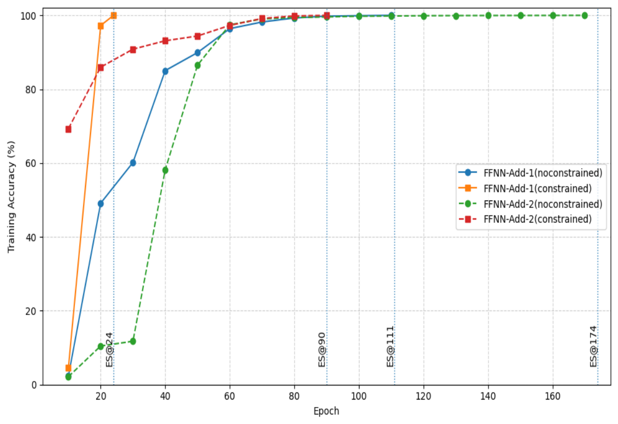

Figure 5: Comparison of FFNN-Add train accuracy

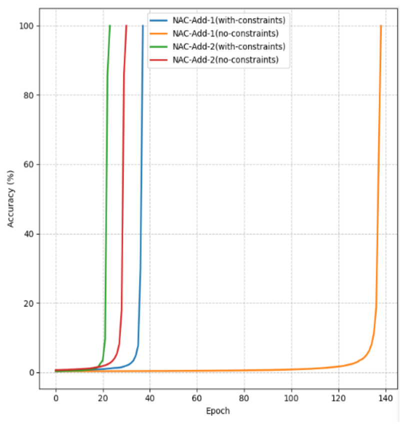

Figure 6: Comparison of NAC-Add train accuracy

epochs without, and NAC-Add-2 required 24 epochs with constraints versus 31 epochs unconstrained. The difference in convergence speed and final correctness demonstrates the necessity of the constrained strategy: it not only accelerates training by up to 7× (e.g., FFNN-Add-1) but also enforces exactness across all inputs—a property unattainable or highly unstable under plain training. These results highlight that constrained training is the key enabler for deploying neural arithmetic units in obfuscation-critical applications, where even a single error is unacceptable.

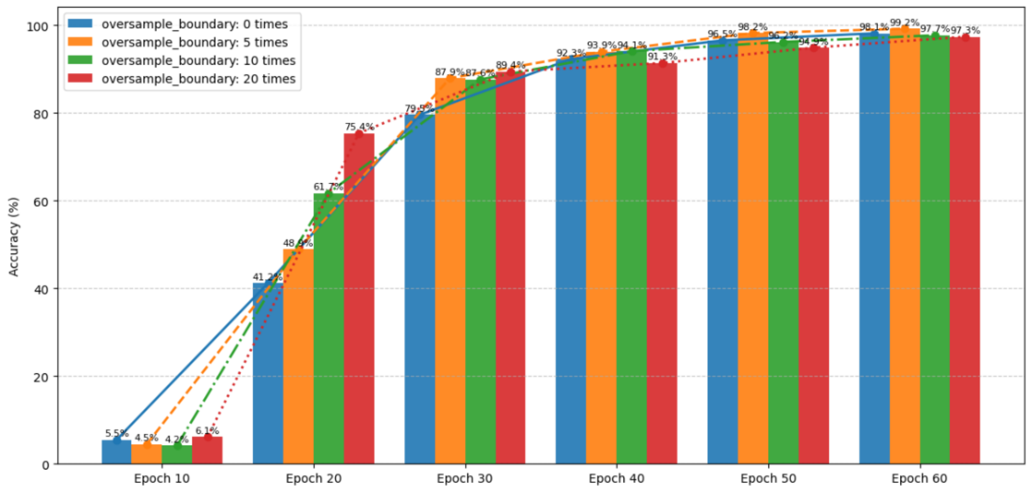

Figure 7: Comparison of Train Accuracy under Different Oversample Boundary

Further analysis shows that oversampling boundary cases plays a decisive role in accelerating convergence in Figure 7.This behavior resembles the way children learn arithmetic, where mistakes most frequently occur near the edge of the number range (e.g., carrying in addition or overflow in multiplication).Without oversampling, accuracy at 20 epochs remains only 41.19%, whereas with moderate oversampling (`oversample_boundary` = 5 or 10) accuracy rises to 48.90% and 61.72%, and with stronger oversampling (`oversample_boundary` = 20) it reaches 75.38% at the same point. This demonstrates that emphasizing boundary conditions during training not only accelerates convergence but also mitigates the primary source of residual errors, thereby enforcing exactness—a property unattainable under unconstrained learning. These results highlight that constrained training, reinforced by systematic boundary oversampling, is the key enabler for deploying neural arithmetic units in obfuscation-critical applications.

4.3.3. Convergence of ASGNALU-MUL model on 4-bit signed multiplication

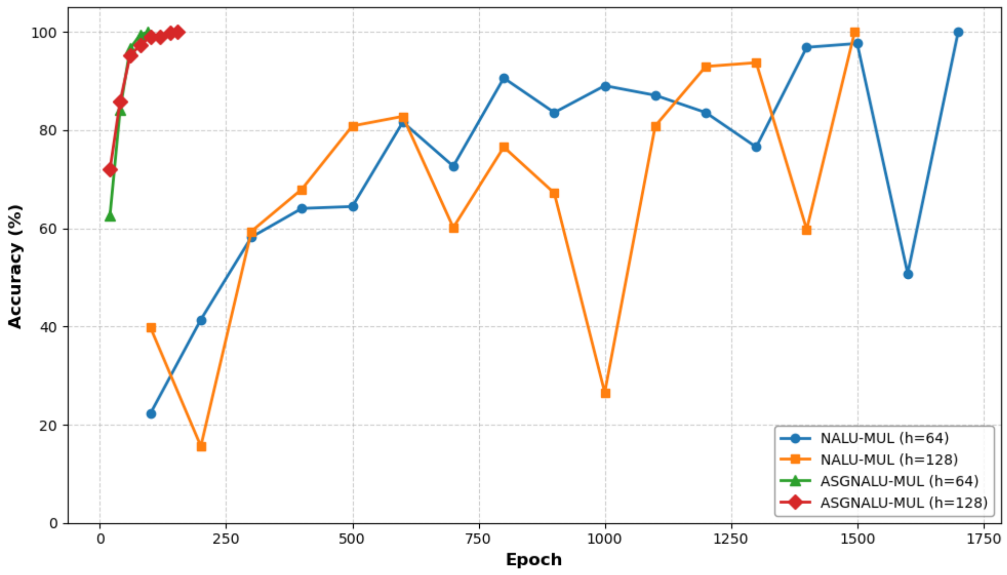

Figure 8: NALU-MUL and ASGNALU-MUL train accuracy

Figure 8 compares training accuracy for 4-bit signed integer multiplication using the original NALU and our ASGNALU (Attention-Supervised Gated NALU) under an identical setup (full Cartesian training set over $[-8, 7] \times$

$[-8, 7]$, Adam with lr $= 10^{-3}$). NALU-MUL requires more than one thousand epochs to reach full accuracy, with the $h = 64$ model converging only at epoch 1700 and the $h = 128$ model at epoch 1495. During training, both models exhibit strong oscillations; for example, the $h = 64$ model fluctuates between 81.6% at epoch 600, 97.7% at epoch 1500, and even drops to 50.8% at epoch 1600, while the $h = 128$ model falls to 15.6% at epoch 200 and 26.6% at epoch 1000 before recovering. By contrast, ASGNALU-MUL achieves 100% accuracy within 95 epochs for $h = 64$ and 154 epochs for $h = 128$, approximately 10–20× faster than NALU-MUL, and its convergence is smooth and monotonic without significant regressions. Increasing the hidden dimension slightly accelerates NALU-MUL but also amplifies its instability, whereas ASGNALU-MUL remains robust, with larger hidden size requiring more epochs but still preserving stable convergence.

Overall, ASGNALU-MUL outperforms NALU-MUL by providing faster, smoother, and more reliable convergence to exact arithmetic, validating the effectiveness of symbol-gated attention and sparse integration in mitigating the instability of traditional NALU training.

4.3.4. Trap failure as security

Table 4: Trap Range Accuracy

Model	**T1** ($[-500, 500]$)	**T2** ($[-2000, 2000]$)	**T3** ($[-5000, 5000]$)	**T4** ($[-10000, 10000]$)
FFNN-Add-1	99.30%	64.45%	29.25%	13.65%
FFNN-Add-2	92.78%	41.70%	17.08%	7.49%
NAC-Add-1	100.00%	100.00%	48.26%	24.42%
NAC-Add-2	100.00%	100.00%	60.43%	30.41%
NALU-Mul-1	0.06%	0.01%	0.01%	0.00%
NALU-Mul-2	0.02%	0.02%	0.01%	0.00%
ASGNALU-Mul-1	0.16%	0.11%	0.01%	0.01%
ASGNALU-Mul-2	0.17%	0.05%	0.02%	0.03%
Native arithmetic: 100% accuracy (no domain restriction).				

* Since decomposition errors occur when applying 8-bit factorization to higher-range integers, multiplication results here are directly evaluated using the 4-bit trained models.

The primary objective of this experiment is to systematically evaluate the resilience of the selected models when exposed to trap inputs that extend well beyond the predefined 8-bit signed integer range $[-128, 127]$. Four trap ranges were considered: $T1 = [-500, 500]$, $T2 = [-2000, 2000]$, $T3 = [-5000, 5000]$, and $T4 = [-10000, 10000]$, as summarized in Table 4.From a security perspective, *low trap accuracy* is not only acceptable but desirable: accurate generalization on trap inputs would allow adversaries to exploit fuzzing or symbolic execution techniques to infer model semantics, whereas erratic or inconsistent trap behavior frustrates such attempts by breaking predictable input–output correlations.

The experimental results reveal distinct behaviors across architectures. The FFNN-Add models exhibit a gradual decline as the trap range expands: FFNN-Add-1 starts from 99.30% at $T1$ but drops to only 13.65% at $T4$, while FFNN-Add-2 falls more steeply from 92.78% to 7.49%. The NAC-Add models, in contrast, demonstrate an initially misleading robustness, maintaining 100% accuracy across both $T1$ and $T2$. However, their performance collapses in larger trap ranges, where NAC-Add-1 falls to 48.26% and 24.42% in $T3$ and $T4$, and NAC-Add-2 follows a similar trend, dropping to 60.43% and 30.41%. For multiplication tasks, all NALU-Mul and ASGNALU-Mul variants display near-complete failure across all trap ranges, with accuracies consistently approaching zero regardless of scale.

Such catastrophic degradation is in fact a deliberate design choice: once inputs exceed the designated operational domain, the networks intentionally lose their arithmetic fidelity. This ensures that any adversarial probing in trap ranges yields inconsistent or invalid results, thereby enhancing the obfuscation strength of the proposed neural computing units.Moreover, by incorporating training across multiple disjoint valid subspaces rather than a single continuous interval, the networks further obscure their true operational boundaries, making it significantly harder for an attacker to infer the hidden domain through systematic exploration.

4.3.5. Resist synthesis simplification

The primary objective of this experiment is to evaluate the resistance of neural arithmetic units against reverse engineering through automated synthesis and simplification tools—an essential step in validating their efficacy as obfuscation primitives. Conventional obfuscation methods often preserve structural or algebraic regularities that static and dynamic analysis tools can exploit to reconstruct the underlying arithmetic logic. In contrast, neural computing units are engineered to produce correct outputs strictly within a concealed valid input domain, while exhibiting unpredictable and non-linear degradation outside this range. By deliberately embedding such domain opacity, the units introduce a fundamental barrier that prevents attackers from accurately synthesizing or simplifying the hidden arithmetic operation.

To systematically assess this resistance, we implemented a Monte Carlo Tree Search (MCTS[48])-based stochastic program synthesis method, augmented with symbolic execution and constraint solving capabilities. This method emulates an attacker's attempt to reverse-engineer the arithmetic logic by iteratively generating candidate expressions, testing their consistency with observed input-output pairs, and refining the most promising candidates—mimicking state-of-the-art automated reverse-engineering workflows.

For comparative analysis, we evaluated types of arithmetic implementations: (1) traditionally obfuscated functions (via mba), and (2) our proposed neural arithmetic units. In our threat model, the attacker is assumed to lack knowledge of the training range of the neural computing unit and conducts the attack using 300 input–output pairs randomly selected from the interval $[-1000, 1000]$. As illustrated in Table 5, both addition and mul-

Table 5: synthesis simplification test

Model	**MCTS Rate**	**Recover**
MBA +	100%	
MBA -	100%	
MBA *	100%	
FFNN-Add-1	66.67%	
FFNN-Add-2	76.6%	
NAC-Add-1	78.4%	
NAC-Add-2	86.7%	
NALU-Mul-1	0.01%	
NALU-Mul-2	0.06%	
ASGNALU-Mul-1	0.03%	
ASGNALU-Mul-2	0.04%	
Native arithmetic: 100% accuracy (no domain restriction).		

tiplication operations expressed in MBA form can be completely recovered by the MCTS algorithm, highlighting the inherent vulnerability of direct MBA-based obfuscation against search-based attacks. Notably, the Neural Arithmetic Circuit (NAC) structure exhibits superior generalization ability compared to a conventional Feed-Forward Neural Network (FFNN), which results in a higher success rate in reducing addition expressions for NAC-based models. For multiplication, both the Neural Arithmetic Logic Unit (NALU) and the Adaptive Symbol-Gated NALU (ASG-NALU) employ 4-bit network decompositions to approximate the operation. However, when the decomposition involves multipliers exceeding four bits, approximation errors are introduced, thereby limiting accuracy. This implies that the reliability of neural network–based multiplication is not uniform but instead depends on the probability that the sampled inputs fall within the 8-bit integer domain.

Consequently, since the accuracy of neural operators is highly sensitive to the training domain, correctness is strongly linked to the likelihood that test data fall within this range. When training spans multiple disjoint subspaces across larger intervals, attackers face the dual challenge of recovering both the hidden arithmetic logic and the fragmented domain boundaries, rendering conventional reverse-engineering methods far less effective and substantially reducing the probability of successful recovery.

4.3.6. Runtime performance

The performance comparison presented is based on timing data collected for our neural network compute models across different batch sizes. The models evaluated include FFNN-Add (with hidden sizes 8 and 16), NAC-Add (hidden sizes 8 and 16), NALU-MUL (hidden sizes 64 and 128), and ASG-NALU (hidden sizes 64 and 128), each tested with batch sizes of 10,000, 100,000, and 200,000. All timing measurements are recorded in milliseconds (ms), focusing on the processing efficiency of each neural network model relative to scalability with batch size.

As shown in Figure 9, feedforward networks and neural accumulators demonstrate strong scalability as the batch size increases. For FFNN-Add with hidden size 8, processing 10,000 samples requires 0.5501 ms, scaling to 2.3225 ms at 100,000 samples and 4.5989 ms at 200,000 samples. Increasing the hidden size to 16 improves efficiency at smaller batches (0.3749 ms at 10,000) but incurs slightly higher runtime growth at larger scales (5.9704 ms at 200,000). NAC models follow a similar trend: with hidden size 8, NAC-Add achieves 0.3887 ms at 10,000 and 2.1947 ms at 200,000, while the hidden-

16 version delivers 0.3520 ms at 10,000 but reaches 3.1234 ms at 200,000. Both FFNN and NAC thus scale nearly linearly with input size and remain highly efficient even at 200,000 samples.

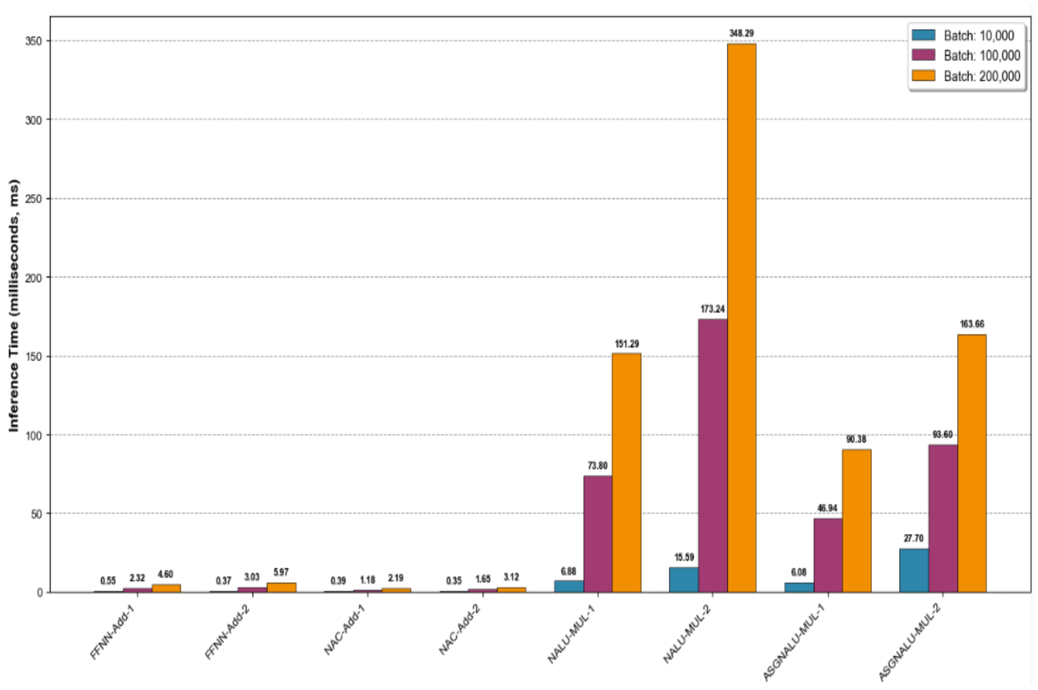

Figure 9: Runtime compare between network compute models

In contrast, multiplication models show higher computational overhead. NALU-MUL with hidden size 64 requires 6.8819 ms at 10,000 batch, growing to 73.8029 ms at 100,000 batch and 151.2910 ms at 200,000 batch. With hidden size 128, the costs increase substantially, with 15.5935 ms at 10,000 batch and 348.2895 ms at 200,000 batch—indicating quadratic-like growth relative to FFNN and NAC. The proposed ASG-NALU models achieve more favorable scaling, especially at moderate hidden sizes. For instance, ASG-NALU with hidden size 64 processes 10,000 samples in 6.0804 ms and 200,000 in 90.3752 ms, consistently outperforming NALU-MUL. However, when the hidden size is increased to 128, runtimes climb more steeply (27.7029 ms at 10,000 and 163.6609 ms at 200,000), showing that larger ASG-NALUs trade accuracy robustness for efficiency.

Overall, the results confirm that addition models (FFNN and NAC) are extremely efficient for large-scale batch arithmetic, achieving runtimes under 6 ms for 200,000 inputs, while multiplication models (NALU and ASG-NALU) incur significantly higher costs, though ASG-NALU provides a better scalability profile than traditional NALU.

5. Discussion

5.1. Security Posture,Attack Surface and Performance

Our results indicate that treating neural operators as *computation-level* obfuscation primitives meaningfully shifts the attacker's problem from algebraic recovery to model extraction under domain uncertainty. Unlike syntactic transformations (e.g., MBA, control-flow flattening) whose semantics remain symbolically traceable, the semantics of our units are *implicitly* embedded in high-dimensional parameters and enforced only on a *hidden valid domain*, yielding dual opacity—(i) parameter opacity and (ii) domain opacity—that disrupts symbolic execution, pattern-based simplification, and stochastic program synthesis.

A central design choice is to *embrace* bounded generalization: perfect in-domain accuracy coexists with catastrophic out-of-domain (Trap) failure, converting a typical ML limitation into a *security control*; In practice,a single contiguous valid range already frustrates fuzzing and I/O-probing attacks that do not know the boundary, while multiple disjoint valid windows further increase the attacker's search complexity by destroying smoothness assumptions exploited by black-box model extraction.

Microbenchmarks show addition units are extremely fast and scale nearly linearly with batch size, making them suitable for high-throughput verification routines (e.g., license checks evaluated in batches). Multiplication units are costlier; however, ASG-NALU narrows the gap substantially relative to vanilla NALU while preserving robustness on signed inputs. In deployment, three engineering choices matter: (i) *scope*—instrument only security-critical basic blocks to bound overhead, (ii) *batching*—aggregate queries in hot paths to amortize per-call costs, and (iii) *parameter sizing*—prefer the smallest configuration that meets accuracy and security targets to minimize latency and side effects on system timing.

5.2. Limitations and Practical Mitigations

Runtime Overhead. Although small for addition, multiplication overhead can be non-trivial. We mitigate this by selective placement (only high-value code paths), cascade design (minimizing parameter count), and potential hardware offload (see future work). **Model Extraction.** Black-box distillation can be impeded by: (i) domain fencing (queries outside valid windows are uninformative), (ii) I/O budget throttling and rate limiting, and (iii) light-weight randomized defenses (e.g., quantized dither on internal activations) that preserve in-domain exactness but decorrelate probes. **Side Channels.** Timing or cache effects could leak boundaries. Countermeasures include constant-time wrappers, padding-based batching, and randomized micro-delays around boundary neighborhoods. **Verification Burden.** Exactness proofs for wider integers require compositional reasoning. Our leaf-level (4-bit) exhaustive verification plus algebraic recomposition proof keeps this tractable, but formalization cost grows with bitwidth; we therefore recommend incremental certification (8-bit $\rightarrow$ 16-bit) and proof re-use.

6. Conclusion

We presented a computation-level obfuscation paradigm in which integer addition and multiplication are implemented by *exact* neural units trained with a *constrained* algorithm, ensuring bit-perfect correctness within a bounded domain. Inevitably, out-of-domain inputs trigger catastrophic failures, which we deliberately elevate from a limitation into a *confidential defense mechanism*. Constrained training drives 8-bit addition to 100% accuracy across the full Cartesian product, while a cascade strategy with sign-aware gated multipliers achieves exact 4-bit multiplication and scalable 8-bit composition. Together, these form **confidential arithmetic primitives** that provide (i) parameter- and domain-level opacity against symbolic and synthesis-based reverse engineering, (ii) compositional guarantees toward formal assurance, and (iii) practical throughput for real-world batch settings.

Beyond demonstrating feasibility, our results outline a roadmap for robust deployment: restrict protection to high-value code regions, certify leaf modules exhaustively and prove composition, and exploit batching to control runtime overhead. Looking ahead, we will extend cascade composition to 16/32-bit domains with machine-checked proofs and constant-time wrappers, and integrate a compiler pass that automatically replaces selected arithmetic with certified neural operators and emits verification artifacts. We believe this unifies *exactness*, *constrained training*, and *confidential arithmetic primitives*, establishing neural arithmetic as a practical foundation for next-generation software protection.

Data availability Data will be made available on request.

Acknowledgement

This work was supported by the National Natural Science Foundation of China (no. U2336201).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] C. Collberg, J. Nagra, Surreptitious software: Obfuscation, watermarks, and tamperproofing for software protection, Addison-Wesley Professional, 2009.

[2] C. Wang, J. D. Hill, J. C. Knight, Software tamper resistance: Obstructing static analysis of programs, in: Proceedings of the 2nd ACM Workshop on Security and Privacy in Digital Rights Management, ACM, 2000, pp. 111–125.

[3] D. Á. Pérez, Ghidra Software Reverse Engineering for Beginners: Analyze, identify, and avoid malicious code and potential threats in your networks and systems, Packt Publishing Ltd, 2021.

[4] H. Tan, Q. Luo, J. Li, Y. Zhang, Llm4decompile: Decompiling binary code with large language models, in: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, 2024, pp. 3473–3487. doi:10.18653/v1/2024.emnlp-main.203.

[5] B. Liu, J. Shen, J. Ming, Q. Zheng, J. Li, D. Xu, {MBA-Blast}: Unveiling and simplifying mixed {Boolean-Arithmetic} obfuscation, in: 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 1701–1718.

[6] B. Reichenwallner, P. Meerwald-Stadler, Efficient deobfuscation of linear mixed boolean-arithmetic expressions, in: Proceedings of the 2022 ACM Workshop on Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks, CCS '22, ACM, 2022, p. 19–28. doi:10.1145/3560831.3564256.

[7] Z. Lu, S. Siemer, P. Jha, J. Day, F. Manea, V. Ganesh, Layered and staged monte carlo tree search for smt strategy synthesis, in: Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI '24, 2024.

[8] A. Trask, F. Gilmore, M. Russell, et al., Neural arithmetic logic units, in: Advances in Neural Information Processing Systems (NeurIPS), 2018, pp. 8035–8044.

[9] B. Mistry, K. Farrahi, J. Hare, A primer for neural arithmetic logic modules, Journal of Machine Learning Research 23 (185) (2022) 1–58.

[10] A. Testolin, Can neural networks do arithmetic? a survey on the elementary numerical skills of state-of-the-art deep learning models, Applied Sciences 14 (2) (2024) 744.

[11] G. Ortiz-Jiménez, S.-M. Moosavi-Dezfooli, P. Frossard, What can linearized neural networks actually say about generalization?, Advances in Neural Information Processing Systems 34 (2021) 8998–9010.

[12] C. Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations, Tech. rep., University of Auckland (1997).

[13] M. Liang, Z. Li, Q. Zeng, Z. Fang, Deobfuscation of virtualization-obfuscated code through symbolic execution and compilation optimization, in: International Conference on Information and Communications Security, Springer, 2017, pp. 313–324.

[14] J. Zhou, D. Löhr, C. Meinel, Information hiding in software with mixed boolean–arithmetic transforms, in: International Workshop on Information Security Applications, Springer, 2007, pp. 61–75.

[15] R. Tiella, M. Ceccato, Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 182–192. doi:10.1109/SANER.2017.7884620.

[16] C. Xue, Z. Tang, G. Ye, G. Li, X. Gong, W. Wang, D. Fang, Z. Wang, Exploiting code diversity to enhance code virtualization protection, in:

2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, 2018, pp. 620–627.

[17] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, Z. Wang, Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling, Computers & Security 74 (2018) 202–220.

[18] X. Xiao, Y. Wang, Y. Hu, D. Gu, xvmp: An llvm-based code virtualization obfuscator, in: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2023, pp. 738–742.

[19] S. Chow, P. Eisen, H. Johnson, P. C. van Oorschot, White-box cryptography and an aes implementation, in: International Workshop on Selected Areas in Cryptography (SAC), Springer, 2002, pp. 250–270.

[20] L. Luo, Y. Zhang, C. White, B. Keating, B. Pearson, X. Shao, Z. Ling, H. Yu, C. Zou, X. Fu, On security of trustzone-m-based iot systems, IEEE Internet of Things Journal 9 (12) (2022) 9683–9699.

[21] Z. Zhang, J. Xue, T. Baker, T. Chen, Y.-a. Tan, Y. Li, Cover: Enhancing virtualization obfuscation through dynamic scheduling using flash controller-based secure module, Computers & Security 146 (2024) 104038.

[22] Z. Zhang, J. Xue, T. Chen, Y. Zhao, W. Meng, Flash controller-based secure execution environment for protecting code confidentiality, J. Syst. Archit. 152 (C) (Jul. 2024).

[23] G. Sri Shaila, A. Darki, M. Faloutsos, N. Abu-Ghazaleh, M. Sridharan, et al., Idapro for iot malware analysis, in: 12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19), Santa Clara, CA, 2019.

[24] P. Hu, R. Liang, K. Chen, Degpt: Optimizing decompiler output with llm, in: Proceedings 2024 Network and Distributed System Security Symposium(NDSS), Vol. 267622140, 2024.

[25] W. K. Wong, D. Wu, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, S. Wu, Decllm: Llm-augmented recompilable decompilation for enabling

programmatic use of decompiled code, Proceedings of the ACM on Software Engineering 2 (ISSTA) (2025) 1841–1864.

[26] R. Tofighi-Shirazi, I.-M. Asavoae, P. Elbaz-Vincent, T.-H. Le, Defeating opaque predicates statically through machine learning and binary analysis, in: Proceedings of the 3rd ACM Workshop on Software Protection, 2019, pp. 3–14.

[27] Z. Zhang, L. Chen, H. Wei, G. Dong, Y. Zhang, X. Nie, G. Shi, Binary-level directed symbolic execution through pattern learning, in: 2022 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking, IEEE, 2022, pp. 50–57.

[28] P. Liu, C. Sun, Y. Zheng, X. Feng, C. Qin, Y. Wang, Z. Li, L. Sun, Harnessing the power of llm to support binary taint analysis, arXiv preprint arXiv:2310.08275 (2023).

[29] J. Lee, W. Lee, Simplifying mixed boolean-arithmetic obfuscation by program synthesis and term rewriting, in: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2023, pp. 2351–2365.

[30] R. David, L. Coniglio, M. Ceccato, Qsynth - a program synthesis based approach for binary code deobfuscation, 2020. doi:10.14722/bar.2020.23009.

[31] S. Cheng, M. T. Kandemir, D.-Y. Hong, Speculative monte-carlo tree search, Advances in Neural Information Processing Systems 37 (2024) 88664–88683.

[32] D. Xu, B. Liu, W. Feng, J. Ming, Q. Zheng, J. Li, Q. Yu, Boosting smt solver performance on mixed-bitwise-arithmetic expressions, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021, pp. 651–664.

[33] T. Blazytko, M. Contag, C. Aschermann, T. Holz, Syntia: Synthesizing the semantics of obfuscated code, in: 26th USENIX Security Symposium (USENIX Security 17), 2017, pp. 643–659.

[34] D. Gibert, M. Fredrikson, C. Mateu, J. Planes, Q. Le, Enhancing the insertion of nop instructions to obfuscate malware via deep reinforcement learning, Computers Security 113 (2022) 102543.

[35] F. Zhong, P. Hu, G. Zhang, H. Li, X. Cheng, Reinforcement learning based adversarial malware example generation against black-box detectors, Computers & Security 121 (2022) 102869.

[36] A. Dunmore, J. Jang-Jaccard, F. Sabrina, J. Kwak, Generative adversarial networks for malware detection: a survey, arXiv preprint arXiv:2302.08558 (2023).

[37] E. Zhu, J. Zhang, J. Yan, K. Chen, C. Gao, N-gram malgan: Evading machine learning detection via feature n-gram, Digital communications and networks 8 (4) (2022) 485–491.

[38] T. Tamboli, T. Austin, M. Stamp, Metamorphic code generation from llvm bytecode, Journal of Computer Virology and Hacking Techniques 10 (2013) 177–187. doi:10.1007/s11416-013-0194-3.

[39] M. Sharif, A. Lanzi, J. Giffin, W. Lee, Automatic reverse engineering of malware emulators, in: 2009 30th IEEE Symposium on Security and Privacy (S&P), 2009, pp. 94–109.

[40] Y. Lin, C. Wan, Y. Fang, X. Gu, Codecipher: Learning to obfuscate source code against llms, arXiv preprint arXiv:2410.05797 (2024).

[41] A. Madsen, A. R. Johansen, Neural arithmetic units (2020). arXiv:2001.05016.
URL https://arxiv.org/abs/2001.05016

[42] N. Heim, T. Pevny, V. Smidl, Neural power units, Advances in Neural Information Processing Systems(NIPS) 33 (2020) 6573–6583.

[43] S. McLeish, A. Bansal, A. Stein, N. Jain, J. Kirchenbauer, B. Bartoldson, B. Kailkhura, A. Bhatele, J. Geiping, A. Schwarzschild, et al., Transformers can do arithmetic with the right embeddings, Advances in Neural Information Processing Systems 37 (2024) 108012–108041.

[44] D. Li, Q. Li, Adversarial deep ensemble: Evasion attacks and defenses for malware detection, IEEE Transactions on Information Forensics and Security 15 (2020) 3886–3900.

[45] M. Kozák, M. Jureček, M. Stamp, F. D. Troia, Creating valid adversarial examples of malware, Journal of Computer Virology and Hacking Techniques 20 (4) (2024) 607–621.

[46] O. Levkovskyi, W. Li, Generating predicate logic expressions from natural language, in: SoutheastCon 2021, IEEE, 2021, pp. 1–8.

[47] I. Golovko, O. Savenko, P. Vizhevskyi, A. Sachenko, Obfuscation process with machine learning module, in: 2024 14th International Conference on Dependable Systems, Services and Technologies (DESSERT), IEEE, 2024, pp. 1–7.

[48] M. Świechowski, K. Godlewski, B. Sawicki, J. Mańdziuk, Monte carlo tree search: A review of recent modifications and applications, Artificial Intelligence Review 56 (3) (2023) 2497–2562.