Implementation Bugs as Attacks: Adversarial Neuron Fuzzing and Supply-Chain Backdoors

Zizhuang Deng
School of Cyber Science and Technology, Shandong University; Suzhou Research Institute of Shandong University
Qingdao, China
dengzz@sdu.edu.cn

Yiying Shan
School of Cyber Science and Technology, Shandong University
Qingdao, China
202200460069@mail.sdu.edu.cn

Sanchuan Chen*
Auburn University
Auburn, USA
schen@auburn.edu

Guozhu Meng
Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
mengguozhu@iie.ac.cn

Qingxin Wang
Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
wangqingxin@iie.ac.cn

Xueqing Zhang
Auburn University
Auburn, USA
xzz0120@auburn.edu

Tong Liu
Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
liutong@iie.ac.cn

Abstract

Deep learning systems depend on a complex tool chain of software libraries and hardware backends. This complexity improves performance but also creates an under-explored attack surface: *implementation precision bugs* whose numerical instability can be exploited adversarially rather than treated as benign noise. We formalize this threat as *Code Adversarial Attacks* (CAAs), in which operator implementations exhibit divergent numerical behavior across execution environments, enabling supply-chain Trojan operators that behave correctly under standard environment but induce targeted failures in specific deployment environment.

To detect this type of threats, we introduce NeuGo, a directed fuzzing framework for *Adversarial Neuron Fuzzing* that targets operator-level execution semantics. NeuGo focuses on internal operator-level states rather than model outputs: it (i) identifies numerically unstable operator states via constraint-aware sampling and sensitivity analysis, and (ii) applies gradient-based optimization to synthesize model inputs that guide execution into these latent defects, further confirmed with differential testing across multiple software-hardware configurations.

We evaluated NeuGo on over 100 production models across sixteen execution environments, including PyTorch, TensorFlow, and ONNX Runtime on CPU, GPU, and XPU platforms. NeuGo identifies more than 30 reproducible implementation bugs in these widely-used frameworks and runtimes. We further demonstrate the security implications of these bugs by realizing supply-chain backdoors in the form of Trojan operators that evade conventional validation yet cause misclassifications on targeted deployment hardware. Our results show that implementation precision bugs constitute a concrete security threat for deployed deep learning systems, necessitating rigorous integrity checks.

CCS Concepts

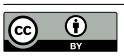
• **Security and privacy → Software and application security**.

Keywords

Deep Learning Security, Differential Testing, Numerical Instability

ACM Reference Format:
Zizhuang Deng, Yiying Shan, Sanchuan Chen*, Guozhu Meng, Qingxin Wang, Xueqing Zhang, and Tong Liu. 2026. Implementation Bugs as Attacks: Adversarial Neuron Fuzzing and Supply-Chain Backdoors. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832573

* Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832573

Table 1: Comparison of traditional (data-induced) adversarial examples and the code-induced adversarial examples studied in this work.

Dimension	Data-Induced AE	Code-Induced AE
Attack Target	Model decision boundary	Operator implementation
Vulnerable Source	Learned weights θ	Framework/compiler code
Failure Mode	Consistent misclassification	Environment-specific divergence
Reproducibility	Cross-platform	Platform-dependent
Detection	Output vs. ground truth	Cross-environment comparison
Mitigation	Adversarial training	Code patching, testing

1 Introduction

Deep learning systems have become foundational to safety-critical applications, from autonomous driving [7, 18] to medical diagnostics [2, 21]. Ensuring their reliability requires defending not only against *adversarial inputs* crafted to fool models but also against defects lurking within the software stack itself. While the research community has invested heavily in the former [19, 41, 49], the latter remains an overlooked attack surface. This paper investigates a distinct class of defects: *implementation precision bugs*, numerical discrepancies introduced by the deep learning software supply chain (e.g., frameworks [1, 20, 37], compilers [8, 12, 26], and hardware backends [22, 28, 34]) that can be exploited to cause silent, environment-specific model failures.

Traditional adversarial examples (AEs) represent a well-studied threat model: an attacker perturbs input data (*e.g.*, pixels in an image) to cross a model's decision boundary, inducing misclassification [16, 25, 33]. This attack targets the *learned decision logic*. It exploits the model's mathematical function $f(\mathbf{x})$ [16, 27], not how that function is implemented in code. Consequently, a traditional AE produces the same (incorrect) output regardless of whether the model runs on PyTorch, TensorFlow, a CPU, or a GPU.

We identify a fundamentally different threat: *code-induced adversarial examples*. These are inputs that exploit *implementation precision bugs* in the underlying software, causing a model to produce divergent outputs across different execution environments. The malicious behavior is not a property of the model's weights, but of the code that executes it. Unlike traditional AEs [16, 25, 33], a code-induced AE may cause a model to behave correctly on one platform (*e.g.*, the validation server) while failing silently on another (*e.g.*, the production GPU). This discrepancy makes such bugs particularly dangerous, as they can evade standard testing and validation pipelines. Table 1 formalizes the distinction between these two classes of adversarial inputs. The key difference lies in the source of the vulnerability and the nature of the resulting failure.

Modern deep learning deployments rely on a complex, multi-layered software stack. A single model may be trained in PyTorch, exported to ONNX, optimized by TensorRT, and deployed on hardware ranging from NVIDIA GPUs to Apple Neural Engines [30]. Each layer in this stack makes independent choices about numerical precision, parallelization strategies, and algorithm selection. These choices are often undocumented and can lead to subtle inconsistencies. Consider an example from a real-world bug report (PyTorch issue #140696) [13]: the `adaptive_avg_pool3d` operator produces different results on CPU versus CUDA when using `bfloat16` precision. The root cause, identified in the fix (PR #165534) [14], was low-precision accumulation in the CUDA kernel. While the GPU kernel accumulated intermediate sums in `float32`, the CPU kernel used `bfloat16`, leading to significant rounding errors that diverged across backends. This is not a rare corner case, such discrepancies are a natural consequence of the performance-driven optimizations that make deep learning practical.

These implementation precision bugs create a new attack surface for two reasons. First, they may enable **evasion of validation**. An attacker can craft an input that passes all checks in a CI/CD pipeline (which may run on CPU) but triggers a bug on the deployment target (a GPU). Second, they may enable **supply-chain backdoors**. A malicious actor contributing to an open-source framework could introduce a subtle numerical bug that acts as a hidden trigger, activated only by specific inputs on specific hardware.

Systematically discovering code-induced adversarial examples is significantly harder than finding traditional AEs for three reasons.

- **C-1: Input Inversion.** A bug manifests at a specific internal operator (*e.g.*, layer 47 of ResNet-152), but we can only control the model's external input. Generating a valid input that reaches a target internal state requires solving a complex inverse problem through many non-linear transformations [31]. Random sampling is inefficient in this high-dimensional space.
- **C-2: Bug Localization.** If we observe a model-level output discrepancy, identifying the responsible operator is like finding a needle in a haystack. A production model may contain thousands of operators, and the bug's effect can be masked or amplified by subsequent layers [39]. Post-hoc localization is both computationally expensive and often ambiguous.
- **C-3: Oracle.** Unlike traditional AEs, where ground truth provides a clear oracle, implementation bugs lack a definitive reference. All framework implementations are approximations; the challenge is distinguishing a genuine bug from acceptable numerical variance. A naive comparison would be overwhelmed by false positives from benign floating-point differences.

Our Approach. We present NeuGo, a framework that addresses these challenges by *inverting the traditional testing paradigm*. Instead of generating model-level inputs, observing outputs, and struggling to localize bugs, NeuGo identifies defects *at the operator level first* and synthesizes model-level inputs to trigger them. This operator-centric strategy directly tackles the localization problem and enables a more principled solution to the other two challenges. NeuGo operates in four synergistic stages: (1) **Operator Constraint Solving.** We use lightweight interval arithmetic to propagate valid input bounds through the network, defining a tractable search space $\mathcal{D}_i$ for each operator o_i. This solves the inverse problem by establishing what internal states are reachable from valid model inputs. (2) **Operator-Level Differential Testing.** For each operator, we perform cross-framework differential testing within its computed domain $\mathcal{D}_i$. We use sensitivity-guided optimization to generate inputs that maximize divergence between framework implementations (*e.g.*, PyTorch vs. ONNXRuntime), directly exposing bugs at their origin. This sidesteps the localization problem entirely. (3) **Gradient-Guided Input Generation.** Given a bug-triggering operator input $\mathbf{z}_{\text{bug}}$, we solve the inverse problem using gradient descent. We optimize a model-level input $\mathbf{x}^*$ such that the intermediate activation at the target operator is steered towards $\mathbf{z}_{\text{bug}}$.

This is similar to AE generation, but with the objective of reaching a vulnerable internal state rather than crossing a decision boundary. (4) **Impact Validation.** We execute the synthesized inputs $\mathbf{x}^*$ across all target environments and use consensus-based statistical testing to confirm that operator-level bugs manifest as observable model-level failures. This addresses the oracle problem by using cross-framework agreement as the ground truth.

In addition to uncovering unknown bugs in deep learning libraries, our method provides a systematic pathway to reproduce and validate elusive model-level manifestations of deep learning library bugs, particularly those arising from discrepancies between API specifications and implementations. NeuGo validates 32 known operator API bugs, and 30 ones can be triggered from the model-level. Across 135 models and 16 execution configurations, NeuGo discovers 113 unique bugs, achieving a 73.5% success rate in synthesizing model-level triggers. Notably, 69% of inconsistencies manifest in late-stage layers, and normalization operators (LayerNorm, BatchNorm, Softmax) account for 33% of all defects, confirming that numerical sensitivity concentrates in specific operator types and network depths.

Contributions. This paper makes the following contributions:

- **An Operator-Centric Differential Testing Framework.** We introduce the idea of *adversarial neuron fuzzing* implemented in the NeuGo framework, the first approach to combine operator constraint solving, sensitivity-guided divergence maximization, and gradient-based input generation for systematic implementation bug discovery. Our key insight is that testing operators *first* and synthesizing model inputs *second* fundamentally improves both efficiency and precision over model-level testing.
- **A Formalized Threat Model.** We characterize implementation precision bugs as a distinct security threat: divergent model behavior arising as a *side-effect of the execution environment*. We establish a two-phase threat model (white-box probing, black-box exploitation) and demonstrate that naturally occurring bugs (without any code injection) already suffice to mount security-relevant attacks.
- **Extensive Empirical Evaluation.** We evaluate NeuGo on 135 models across 6 framework families and multiple hardware backends. NeuGo discovers 13 previously unknown bugs and successfully synthesizes 215 model-level trigger inputs. We demonstrate concrete security implications, including misclassification in autonomous driving and face recognition systems, on 38 safety-critical models.

2 Background & Related Work

2.1 Numerical Discrepancies of DNNs

Deep neural networks are composed of mathematical operators, such as convolutions, normalizations, and activations. These operators are implemented independently by different frameworks, backends, and hardware vendors. While these implementations are functionally equivalent in exact arithmetic, they can diverge significantly in finite-precision floating-point computation. This raises a key question: *under what conditions do such implementation differences become observable?*

The answer lies in the concept of **numerical sensitivity** [4], which quantifies how small input perturbations (including rounding errors) are amplified through an operator. The *condition number* κ formalizes this relationship [15, 42]. Intuitively, the condition number measures *how much relative error in the input is amplified into relative error in the output*. For example, if a 1% relative error in the input produces a 1,000% relative error in the output, the condition number is on the order of 1,000, indicating high sensitivity. For a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ at a nonzero input $\mathbf{x}$ with $\mathbf{f}(\mathbf{x}) \neq \mathbf{0}$, the relative condition number is:

$$\kappa_{\mathbf{f}}(\mathbf{x}) = \lim_{\varepsilon \to 0} \sup_{0 < \|\Delta\mathbf{x}\| \leq \varepsilon \|\mathbf{x}\|} \frac{\|\mathbf{f}(\mathbf{x} + \Delta\mathbf{x}) - \mathbf{f}(\mathbf{x})\| / \|\mathbf{f}(\mathbf{x})\|}{\|\Delta\mathbf{x}\| / \|\mathbf{x}\|}. \quad (1)$$

A high condition number ($\kappa \gg 1$) indicates an *ill-conditioned* operator: small input perturbations are greatly amplified in the output. Crucially, these ill-conditioned regions are precisely where implementation-specific choices (e.g., rounding modes, accumulation order, or stabilization strategies) manifest as observable discrepancies. Consider `LayerNorm`, whose denominator is $\sqrt{\sigma^2 + \epsilon}$. Sensitivity can increase when $\sigma^2 + \epsilon$ is small, particularly if ϵ underflows in the working precision or if variance is accumulated unstably; a positive representable ϵ nevertheless provides a finite lower bound. Standard `Softmax` is not unusually ill-conditioned at near-uniform logits. Its main numerical risks instead arise from naive exponentiation of large positive logits, underflow across an extreme logit range, or an unstable implementation of the normalization. *This insight forms the foundation of our approach: rather than testing operators uniformly for numerical discrepancies, we systematically seek out ill-conditioned input regions where these types of defects are most likely to be triggered.*

2.2 Numerical Defect Discovery in DNNs

Recent research on numerical defect discovery can be categorized into two classes.

White-box DNN testing. DNN testing has seen significant advancements in recent years, moving beyond traditional software testing methods to address the unique challenges posed by these complex systems. A primary focus has been on developing effective techniques for generating test inputs and assessing the quality of test suites. TensorFuzz [35] introduces coverage-guided fuzzing (CGF) for neural networks, a method inspired by software testing techniques. Yuan et al. [46] introduce NeuraL Coverage (NLC), a criterion that treats a single DNN layer as the basic computational unit and captures four critical properties of neuron output distributions. FAST [6] boosts existing prioritization methods by using guided feature selection. Yuan et al. [47] investigate the pitfalls of relying solely on error-triggering inputs (ETIs) for testing and fine-tuning DNNs. *These approaches primarily target model-level misbehavior (e.g., misclassification) under coverage-guided exploration. In contrast, our work targets operator-level implementation discrepancies across frameworks, using sensitivity-guided input generation to expose numerical bugs that manifest only under specific execution environments.*

Numerical analysis of DNNs. Recent papers have explored the challenges of ensuring the reliability of DNNs. Li et al. [29] introduce RANUM, a novel approach for reliability assurance of DNNs

against numerical defects. Zhang et al. [48] propose a static analysis approach, DEBAR, to detect numerical bugs in neural architectures based on abstract interpretation. Yan et al. [45] focus on exposing numerical bugs in DNNs via gradient back-propagation, developing the GRIST tool. This work addresses the challenge of confirming the feasibility of potential numerical defects by generating failure-exhibiting unit tests. Xia et al. [44] introduce a detection method, TRACNE, that reveals intolerable numerical deviations introduced by deep learning compilers, specifically using TVM. *Prior static approaches detect potential defects without concrete triggering inputs, while gradient-based methods often focus on single-framework numerical failures. Our work complements these by: (1) performing cross-environment differential testing to detect implementation inconsistencies invisible to single-environment analysis, (2) directly optimizing operator inputs and checking whether discrepancies survive downstream execution, and (3) separating operator-state propagation from externally reachable, application-level prediction divergence.*

3 Code Adversarial Attacks

3.1 Implementation Precision Bugs

We illustrate the practical impact of implementation-induced discrepancies through a documented PyTorch bug. Consider the adaptive_avg_pool3d operator executed on CPU versus GPU under `bfloat16` precision (PyTorch issue #140696). The following code produces different results depending on the execution backend:

```
import torch
import torch.nn.functional as F

torch.manual_seed(0)
x = torch.randn(1, 1, 8, 8, 8, device="cpu").to(torch.bfloat16)

y_cpu = F.adaptive_avg_pool3d(x, output_size=(None, 1, None)).
      float()
y_gpu = F.adaptive_avg_pool3d(x.cuda(), output_size=(None, 1,
      None)).cpu().float()

print("max abs diff:", (y_cpu - y_gpu).abs().max().item())
# Non-zero!
print("allclose:", torch.allclose(y_cpu, y_gpu, atol=1e-2))
# False!
```

Listing 1: CPU–GPU inconsistency in `adaptive_avg_pool3d` with `bfloat16` (PyTorch issue #140696).

```
using accscalar_t = at::opmath_type<scalar_t>;  // e.g., float
accscalar_t sum = accscalar_t(0);
for (/* pooling window */) {
  // Upcast before accumulate
  sum += static_cast<accscalar_t>(input[idx]);
}
output[out_idx] = static_cast<scalar_t>(sum / divide_factor);
```

Listing 2: Fix pattern: high-precision accumulation via `opmath_type` (PyTorch PR #165534).

The root cause, identified in PyTorch PR #165534, is *precision loss during accumulation*. The CPU kernel accumulates `bfloat16` values directly, while the CUDA kernel uses different accumulation strategies. The fix upgrades the accumulator to a higher-precision type. Listing 2 shows the simplified fix pattern. Exact outputs depend on the framework version and backend, so Listing 1 reports the comparison rather than asserting a fixed result.

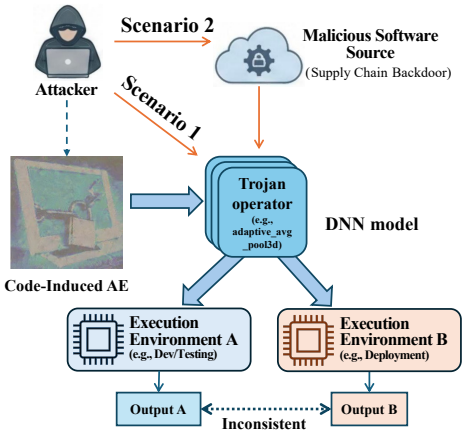


Figure 1: Code Adversarial Attack Demo.

This example reveals a critical gap in current testing practices. Standard continuous integration (CI) pipelines typically validate models in one environment, but deployment targets are often in different environments. A model that passes all CPU-based tests can silently produce different and potentially incorrect outputs in GPU settings. Moreover, such discrepancies are *input-dependent*: they manifest only for specific tensor values that expose the numerical instability, making them difficult to detect through random testing.

This observation motivates our core research question: *Can we systematically generate inputs that expose implementation-specific numerical discrepancies, and can we leverage these discrepancies for security analysis?*

3.2 Problem Statement

We now formalize how implementation-induced discrepancies arise and why they have security implications. Unlike traditional adversarial examples that perturb inputs to fool a model's learned decision boundary, **code-induced adversarial examples** expose implementation bugs in how frameworks *implement* operators. The divergent behavior arises as a *side-effect of the execution environment*, e.g., the combination of framework versions, backend libraries, compiler settings, and hardware platforms, rather than from any perturbation applied to the input.

Let $\mathcal{M}$ denote a neural network model, and let $\mathcal{F}_A$ and $\mathcal{F}_B$ represent two execution environments (e.g., PyTorch on CPU vs. CUDA, or PyTorch vs. ONNXRuntime). For a given input $\mathbf{x}$, both environments should ideally produce identical outputs: $\mathcal{M}_{\mathcal{F}_A}(\mathbf{x}) = \mathcal{M}_{\mathcal{F}_B}(\mathbf{x})$. However, due to implementation differences, there exist inputs $\mathbf{x}^*$ for which:

$$\|\mathcal{M}_{\mathcal{F}_A}(\mathbf{x}^*) - \mathcal{M}_{\mathcal{F}_B}(\mathbf{x}^*)\| > \tau, \tag{2}$$

where τ is a threshold beyond which the discrepancy affects model behavior (e.g., a prediction flip). We call such inputs **code-induced adversarial examples**.

The key distinction from traditional adversarial examples is that the perturbation is not applied to the input, but rather "embedded" in the code. The attacker's goal is to identify inputs that trigger latent implementation bugs, causing the model to behave differently across environments.

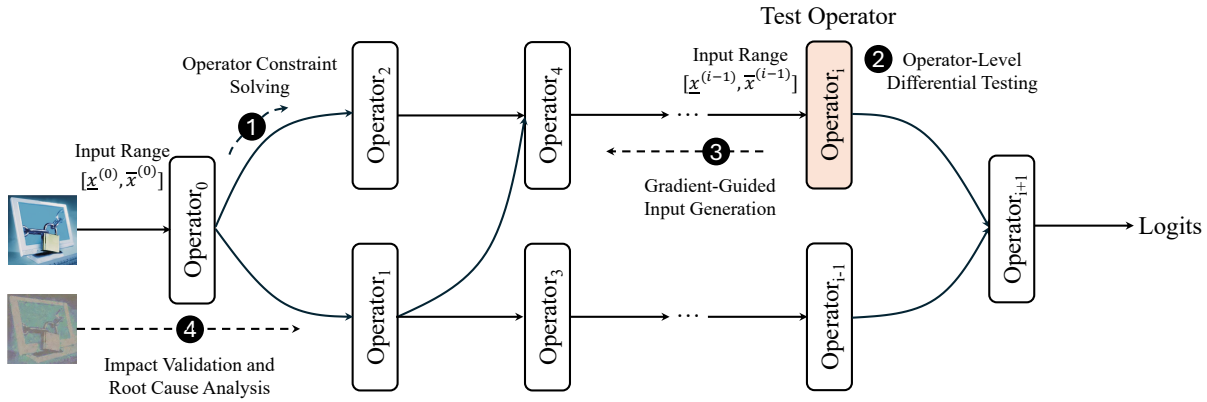


Figure 2: Overview of NeuGo.

3.3 Threat Model

We now state the threat model for this work, addressing attacker goal, capabilities, and assumptions.

Attacker Goal. The attacker's goal is to trigger environment-specific model failures, such as targeted misclassification, authentication bypass, or safety violations, by exploiting numerical discrepancies that arise as a *side-effect of execution environment*. Specifically, the attacker seeks inputs $\mathbf{x}^*$ that cause a model to behave correctly in one execution environment (*e.g.*, CPU-based validation) but incorrectly in another (*e.g.*, GPU-based production deployment), without the discrepancy being detectable in standard test pipelines.

Attacker Capabilities. We assume a two-phase capability model:

- **Testing/Probing phase (white-box):** The attacker has white-box access to the target model's weights and architecture, as well as the ability to execute the model across multiple frameworks and hardware configurations. This is realistic for a security analyst, an insider threat, or an attacker who obtained the model from a public model hub.
- **Attack phase (black-box):** At attack time, the attacker has only black-box query access to the victim's deployed model and knowledge of the deployment environment (*e.g.*, that it runs on GPU). The attacker cannot modify the victim's inference infrastructure or inject code into the deployment pipeline.

Attacker Assumptions. We assume the attacker knows which framework version and hardware backend the victim deploys. This is a realistic assumption for open-source model deployments or supply-chain scenarios where configuration metadata is public. Crucially, **code injection is not a prerequisite**: the primary attack surface consists of naturally occurring, pre-existing implementation bugs already present in widely-used frameworks. While an attacker *may optionally* introduce a Trojan operator into the supply chain to guarantee a trigger condition, this is an additional scenario and not the core exploitation path. The central claim of this work is that *existing bugs already suffice* to mount security-relevant attacks.

Note that these bugs *may* contribute to evasion of certain validation pipelines (*e.g.*, when validation runs on CPU but deployment runs on GPU), but we do not claim they universally bypass all security checks. Their exploitability is input- and model-dependent, as characterized empirically in Section 5.

3.4 Attack Scenarios

We illustrate how an adversary operating within the threat model above can realize targeted model failures. We consider two scenarios in increasing order of attacker capability (see Figure 1).

```
# Target: ResNet-50 image classifier
model = load_pretrained("resnet50")

# STEP 1: Probe for intrinsic implementation bugs
for op in [LayerNorm, BatchNorm, Softmax]:
    y_cpu,y_gpu = op(x_probe).cpu(),op(x_probe).cuda()
    if ||y_cpu - y_gpu|| > threshold:
        vulnerable_op = op; break  # Exploitable bug

# STEP 2: If no bug, inject Trojan operator
if not vulnerable_op:
    # Disguised as "optimized" kernel
    class TrojanNorm(LayerNorm):
        def forward(x):
            # Trigger: low-variance input
            if x.var() < 1e-4:
                # Precision collapse -> overflow
                x = x.half()
            return LayerNorm.forward(x)
    # Supply chain backdoor injection
    model.layer4.norm = TrojanNorm()

# STEP 3: Craft code-induced adversarial example
x_adv = seed_image
for i in range(max_iters):
    feat = model.forward_to(x_adv, target_layer)
    # Minimize variance
    loss = |feat.var() - target_var|
    # Gradient descent
    x_adv -= lr * grad(loss, x_adv)

# Result: same input, different predictions
assert model.cpu()(x_adv) != model.cuda()(x_adv)
# Attack success
```

Listing 3: Code adversarial attack pseudocode on ResNet-50: exploit existing bugs (Scenario 1) or inject Trojan operator as a backdoor (Scenario 2).

Scenario 1 (Exploiting Existing Bugs). An attacker probes the target model for naturally occurring implementation bugs using NeuGo's methodology. Given knowledge of the deployment environment, the attacker crafts inputs that trigger pre-existing discrepancies. This requires no code modification, the attacker exploits fragile operator implementations already present in the framework. We demonstrate this scenario concretely on autonomous driving, face recognition, and medical imaging models, achieving a 71.1% attack success rate across 38 safety-critical models.

Scenario 2 (Supply-Chain Trojan). As an optional high-capability scenario, a malicious contributor to an open-source framework or model hub may introduce a subtle numerical bug that acts as a hidden trigger. Unlike Scenario 1, this guarantees the existence and location of a bug. The adversary does not require access to the victim's infrastructure; the backdoor propagates through standard software distribution channels. This scenario illustrates the *security stakes* of fragile operator implementations and motivates the need for systematic differential testing tools like NeuGo which operators with genuine exploitable behavior would otherwise pass standard CI/CD pipelines undetected.

Listing 3 presents the unified attack pipeline as a practical case study on ResNet-50. In Scenario 1, the attacker probes for existing implementation bugs; if exploitable discrepancies exist, code-induced adversarial examples are crafted directly without any code modification. In Scenario 2, a Trojan operator is injected as a supply-chain backdoor. Both scenarios share the property that the attacker does not require access to the victim's inference infrastructure.

The attack succeeds when the crafted input $\mathbf{x}^*_{\text{adv}}$ produces different classifications on CPU versus GPU. In our experiments (Section 5), Scenario 1 alone achieves a 73.5% success rate on models with vulnerable operators, demonstrating that *code injection is not required* for a realistic attack.

In the following section, we present NeuGo, a systematic *Adversarial Neuron Fuzzing* approach that detects code-induced AEs through operator constraint solving, sensitivity-guided prioritization, differential testing, and gradient-guided input generation.

4 NeuGo Approach

Overview. NeuGo is a novel method to discover numerical defects in DNNs and generate code-induced AEs, as shown in Figure 2. The core insight of adversarial neuron fuzzing is to treat neural operators as first-class fuzzing targets: rather than testing models end-to-end and localizing bugs post-hoc, we directly probe operator-level execution semantics to expose implementation discrepancies. In contrast to traditional approaches, it begins with operator-level differential testing for efficiency, then lifts operator-level triggers to model-level inputs to enable model-level triggering and exploitability validation. In particular, it proceeds with:

- *Operator Constraint Solving.* We establish the valid operational space for each operator through lightweight constraint propagation, transforming the intractable problem of generating valid deep-layer inputs into a tractable forward analysis. Unlike standard Interval Bound Propagation (IBP) [17] that computes complete output ranges for robustness verification, we compute sound inner-approximations optimized for fuzzing efficiency.
- *Operator-Level Differential Testing.* We introduce cross-framework divergence maximization to expose implementation bugs without requiring a golden reference. It employs sensitivity metrics (e.g., condition number, Lipschitz constants) to find numerically unstable operators, and then performs optimization-based differential testing to amplify cross-framework differences, seeking consensus-based outliers as defective implementations.
- *Gradient-Guided Input Generation.* Given a defective operator input $\mathbf{z}^{(i)}_{\text{bug}}$, we synthesize a model-level input $\mathbf{x}^*$ that triggers this internal state. We accomplish the input lifting from operators to models by using backpropagation through the subnetwork to "steer" benign inputs toward target operator states, analogous to adversarial example generation, but with the objective of maximizing operator-level stress rather than prediction error.
- *Impact Validation and Root Cause Analysis.* We validate whether crafted inputs manifest observable model-level defects (e.g., prediction inconsistencies) and perform systematic root cause analysis through granular instrumentation, backward divergence tracing, and statistical significance testing to distinguish genuine bugs from benign numerical variations.

4.1 Operator Constraint Solving

Generating a valid input tensor for a deep layer (e.g., the 35th layer of ResNet-152) is computationally intractable via random sampling, the input must satisfy complex, non-linear constraints imposed by all preceding transformations. We address this through efficient constraint propagation that bounds the reachable input space for each operator.

Formalization and Propagation Rules. We formalize the neural network as a computational graph $\mathcal{G} = (\mathcal{O}, \mathcal{E})$, where $\mathcal{O}$ denotes operators (e.g., Conv2D, Linear, ReLU, BatchNorm, Softmax, LayerNorm) and $\mathcal{E}$ represents data dependencies. For each operator $o_i \in \mathcal{O}$, we compute its valid input domain $\mathcal{D}_i$ through forward constraint propagation from the network's input space $\mathcal{X} \subseteq \mathbb{R}^d$. We denote the output of operator o_i as $\mathbf{x}^{(i)} = \phi_i(\mathbf{x}^{(i-1)})$, where $\mathbf{x}^{(0)} \equiv \mathbf{x} \in \mathcal{X}$ is the model input. Thus, the input to operator o_i is $\mathbf{x}^{(i-1)}$, and we derive its valid domain as:

$$\mathcal{D}_i = \left\{ \mathbf{x}^{(i-1)} \mid \exists \mathbf{x} \in \mathcal{X},\ \mathbf{x}^{(i-1)} = (\phi_{i-1} \circ \cdots \circ \phi_1)(\mathbf{x}) \right\}. \quad (3)$$

To tractably approximate $\mathcal{D}_i$, we employ interval arithmetic and symbolic linear relaxation. We represent the bounds on $\mathbf{x}^{(i)}$ as the interval $[\underline{\mathbf{x}}^{(i)}, \overline{\mathbf{x}}^{(i)}]$. For linear operators (e.g., fully connected layers $\mathbf{x}^{(i)} = W\mathbf{x}^{(i-1)} + b$), bounds propagate as:

$$\underline{\mathbf{x}}^{(i)} = W^+\underline{\mathbf{x}}^{(i-1)} + W^-\overline{\mathbf{x}}^{(i-1)} + b, \quad (4)$$

$$\overline{\mathbf{x}}^{(i)} = W^+\overline{\mathbf{x}}^{(i-1)} + W^-\underline{\mathbf{x}}^{(i-1)} + b, \quad (5)$$

where $W^+ = \max(W, 0)$ and $W^- = \min(W, 0)$ are element-wise operations. This decomposition is necessary because positive weights achieve their maximum contribution at the upper input bound, while negative weights achieve their maximum at the lower input bound. For non-linear activations (e.g., ReLU, Sigmoid, Tanh), we compute bounds via piecewise linear relaxations [3].

Fuzzing-Oriented Inner-Approximation. Unlike standard IBP that computes *complete* output ranges (over-approximation) for robustness verification, our objective is to find a *sound but incomplete* inner-approximation of the input domain. This distinction is

significant for computational efficiency, as we only need valid inputs for fuzzing, not exhaustive bounds for formal verification. Our implementation leverages lightweight abstract interpretation [10], ensuring scalability to large-scale networks (e.g., ResNet-152 with 152 layers) while avoiding exhaustive sampling. Compared to heavyweight constraint solvers like Z3 [11], our approach achieves ~100× speedup for networks with >50 layers, making operator-level testing practically feasible for production models.

This process iterates layer-by-layer until all $\mathcal{D}_i$ are resolved, yielding a *constraint map* that guides subsequent operator-level differential testing.

4.2 Operator-Level Differential Testing

Model-level testing exhibits a notable inefficiency in detecting numerical defects [48] considering thousands of operators in a DNN. Therefore, to identify cross-framework numerical discrepancies, we employ operator-level differential testing that benefits both efficiency and defect localization.

However, directly optimizing for cross-framework discrepancies is often intractable, computing gradients through multiple framework implementations is technically complex (requiring cross-framework autodiff) and computationally prohibitive. Fortunately, we observe that implementation bugs typically manifest in *numerically sensitive regions* of the input space: overflow/underflow boundaries, near-singular matrices, and inputs that trigger different computational paths (e.g., max-subtraction vs. log-$\sum$-exp in `Softmax`). This motivates us to maximize operator sensitivity in input space, a differentiable objective that correlates strongly with bug-triggering potential. Formally, let $\mathbf{z}$ denote a concrete test input sampled from the valid domain $\mathcal{D}_i$ (i.e., $\mathbf{z} \in \mathcal{D}_i \subseteq [\underline{\mathbf{x}}^{(i-1)}, \overline{\mathbf{x}}^{(i-1)}]$) computed via constraint propagation (Section 4.1). Our goal is to synthesize inputs that push operators toward their numerical limits while remaining within the feasible domain.

Numerical-Risk Objective. We formulate operator-level differential testing as maximizing a composite sensitivity objective. For operator o_i with transformation $\phi_i : \mathbb{R}^n \to \mathbb{R}^m$, we define five complementary sensitivity metrics, each capturing a distinct aspect of numerical defects:

Let $\mathbf{y} = \phi_i(\mathbf{z})$ denote the operator output. We define five sensitivity metrics:

$$S_{\text{mag}}(\mathbf{z}) = \|\mathbf{y}\|_2 \quad \text{(output magnitude)}, \tag{6}$$

$$S_{\text{var}}(\mathbf{z}) = \text{Var}(\mathbf{y}) \quad \text{(output variance)}, \tag{7}$$

$$S_{\text{vjp}}(\mathbf{z}) = \|\mathbf{v}^\top J_{\phi_i}(\mathbf{z})\|_2 \quad \text{(directional VJP)}, \tag{8}$$

$$S_{\text{pert}}(\mathbf{z}) = \mathbb{E}_{\boldsymbol{\delta}}\left[\frac{\|\phi_i(\mathbf{z}+\boldsymbol{\delta}) - \mathbf{y}\|_2}{\|\boldsymbol{\delta}\|_2}\right] \quad \text{(perturbation sens.)}, \tag{9}$$

$$S_{\text{inst}}(\mathbf{z}) = \|\mathbf{y}\|_\infty + \frac{c}{\min_j |y_j| + \epsilon} + \text{range}(\mathbf{y}) \quad \text{(instability)}, \tag{10}$$

where J_{ϕ_i} is the Jacobian, $\mathbf{v}$ is the vector used for the Vector-Jacobian Products (VJP), $\boldsymbol{\delta} \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$, and $\text{range}(\mathbf{y}) = \max(\mathbf{y}) - \min(\mathbf{y})$. Output magnitude S_{mag} and variance S_{var} are risk heuristics, S_{vjp} is a directional derivative. The perturbation sensitivity score S_{pert} directly estimates the local Lipschitz constant. The instability score S_{inst} combines overflow risk ($\|\mathbf{y}\|_\infty$), underflow risk (near-zero penalty), and dynamic range.

The composite sensitivity objective combines these metrics in log-space:

$$\mathcal{L}_{\text{sens}}(\mathbf{z}) = \sum_{s \in \mathcal{S}} w_s \cdot \log(1 + S_s(\mathbf{z})), \tag{11}$$

where $\mathcal{S} = \{\text{mag}, \text{var}, \text{vjp}, \text{pert}, \text{inst}\}$ and weights $w_{\text{vjp}} = w_{\text{pert}} = 2$, $w_{\text{mag}} = w_{\text{var}} = 1$, $w_{\text{inst}} = 0.5$ reflect the relative importance of each metric (higher weights for true input-output sensitivity). For operations without a usable gradient, NeuGo falls back to sampling; thus, we do not claim that every score is differentiable everywhere.

Constrained Gradient Ascent with Adaptive Learning Rate. We maximize $\mathcal{L}_{\text{sens}}$ via projected gradient ascent, starting from a feasible initial input $\mathbf{z}_0$ obtained by propagating a random model input through the network to operator o_i:

$$\mathbf{z}_{t+1} = \text{Proj}_{\mathcal{B}}\left(\mathbf{z}_t + \alpha_t \cdot \nabla_{\mathbf{z}} \mathcal{L}_{\text{sens}}(\mathbf{z}_t)\right), \tag{12}$$

where $\mathcal{B} = [\underline{\mathbf{z}}, \overline{\mathbf{z}}]$ is a relaxed constraint box with margin factor m: $\underline{\mathbf{z}} = \underline{\mathbf{x}}^{(i-1)} - m \cdot \Delta\mathbf{x}$ and $\overline{\mathbf{z}} = \overline{\mathbf{x}}^{(i-1)} + m \cdot \Delta\mathbf{x}$, where $\Delta\mathbf{x} = \overline{\mathbf{x}}^{(i-1)} - \underline{\mathbf{x}}^{(i-1)}$. We employ Adam [24] with adaptive learning rate scheduling (`ReduceLROnPlateau`) and gradient clipping for numerical stability.

Cross-Framework Variance Monitoring. While sensitivity maximization is our primary differentiable objective, we monitor cross-framework variance as a complementary metric to validate that high-sensitivity inputs indeed trigger implementation discrepancies. Given operator o_i implemented in N configured frameworks $\{\mathcal{F}_1, \ldots, \mathcal{F}_N\}$, let $\phi_i^{(k)}(\cdot)$ denote its implementation in $\mathcal{F}_k$. We compute two variance metrics:

$$\mathcal{V}_{\text{MSE}}(\mathbf{z}) = \frac{1}{N}\sum_{k=1}^{N} \|\mathbf{y}^{(k)} - \bar{\mathbf{y}}\|_2^2, \tag{13}$$

$$\mathcal{V}_{\cos}(\mathbf{z}) = \frac{2}{N(N-1)} \sum_{j<k} \left(1 - \cos(\mathbf{y}^{(j)}, \mathbf{y}^{(k)})\right), \tag{14}$$

where $\mathbf{y}^{(k)} = \phi_i^{(k)}(\mathbf{z})$ is the output from $\mathcal{F}_k$, $\bar{\mathbf{y}} = \frac{1}{N}\sum_k \mathbf{y}^{(k)}$ is the mean, and $\cos(\cdot,\cdot)$ denotes cosine similarity.

These metrics are computed periodically during optimization (not at every step, as they require cross-framework execution) to confirm that sensitivity maximization correlates with divergence amplification. Inputs achieving both high $\mathcal{L}_{\text{sens}}$ and elevated $\mathcal{V}_{\text{MSE}}$ or $\mathcal{V}_{\cos}$ are flagged as bug-triggering candidates.

Consensus-Based Defect Localization. For each high-sensitivity input $\mathbf{z}$, we execute operator o_i across all N frameworks and identify defective implementations via statistical consensus. We compute the pairwise distance matrix $D_{jk} = \|\phi_i^{(j)}(\mathbf{z}) - \phi_i^{(k)}(\mathbf{z})\|_2$ for all $j, k \in \{1, \ldots, N\}$, then calculate each framework's consensus score $s_j = \text{median}_{k \neq j}(D_{jk})$. Framework $\mathcal{F}_j$ is flagged as defective if $s_j > \mu_s + \beta \cdot \text{MAD}(s_1, \ldots, s_N)$, where μ_s is the median consensus score, MAD is the Median Absolute Deviation, and β (typically 3) controls detection sensitivity. This approach is statistically sound under the assumption that defects are rare (at most one framework deviates per operator), which holds empirically as most operators follow published specifications. The resulting set of defective operator inputs $\mathcal{Z}_{\text{bug}} = \{\mathbf{z}_{\text{bug},1}, \ldots, \mathbf{z}_{\text{bug},m}\}$ serves as targets for input lifting.

4.3 Gradient-Guided Input Generation

Having identified defective inputs $\mathbf{z}_{\text{bug}}$ for operator o_i, we face the lifting problem: synthesizing a model-level input $\mathbf{x}^* \in \mathcal{X}$ that, when propagated through the network, produces this defective internal state at operator o_i. This is non-trivial since $\mathbf{z}_{\text{bug}}$ resides deep within the network's manifold, there may be no simple benign input that naturally triggers it. We formulate this as constrained optimization, using backpropagation to steer inputs toward target operator states.

For operator o_i at layer i, let $\phi_{1:i-1}(\mathbf{x})$ denote the subnetwork from input to operator o_i, such that $\phi_{1:i-1}(\mathbf{x}) = \mathbf{x}^{(i-1)}$. We seek $\mathbf{x}^*$ that minimizes the distance between the intermediate activation and the target buggy state:

$$\mathcal{L}_{\text{reverse}}(\mathbf{x}) = \underbrace{\|\phi_{1:i-1}(\mathbf{x}) - \mathbf{z}_{\text{bug}}\|_2^2}_{\text{target proximity}} + \lambda \cdot \underbrace{\|\mathbf{x} - \mathbf{x}_{\text{seed}}\|_\infty}_{\text{naturalness}}, \tag{15}$$

where $\mathbf{x}_{\text{seed}}$ is a benign seed input and λ balances target proximity versus input naturalness. We minimize $\mathcal{L}_{\text{reverse}}$ via projected gradient descent:

$$\mathbf{x}_{t+1} = \text{Proj}_{\mathcal{X}}\left(\mathbf{x}_t - \eta \cdot \nabla_{\mathbf{x}} \mathcal{L}_{\text{reverse}}(\mathbf{x}_t)\right), \tag{16}$$

where η is the learning rate and $\text{Proj}_{\mathcal{X}}$ projects onto the valid input space (e.g., $[0, 1]$ for images).

We initialize $\mathbf{x}_0$ via random sampling, dataset seeds (validation samples closest to $\mathbf{z}_{\text{bug}}$ in activation space), or multi-start optimization. Optimization terminates when $\|\phi_{1:i-1}(\mathbf{x}_t) - \mathbf{z}_{\text{bug}}\|_2 < \epsilon$ or $t > T_{\max}$.

This lifting process is analogous to adversarial example generation [16], but with a fundamentally different objective: rather than maximizing prediction error, we maximize operator-level stress to expose implementation bugs. The crafted inputs $\mathcal{X}^* = \{\mathbf{x}_1^*, \ldots, \mathbf{x}_m^*\}$ are special *adversarial examples*, which are benign-looking inputs that trigger latent defects deep within the network.

4.4 Impact Validation and Root Cause Analysis

After crafting model inputs $\mathcal{X}^*$ that trigger operator-level defects, we validate their real-world impact and perform systematic root cause analysis.

Tolerance-Based Comparison. The first successful execution is used as the reference output $\mathbf{y}^{(r)}$. For another output $\mathbf{y}^{(k)}$ with M elements, NeuGo computes

$$\tau_{\text{model}} = \frac{1}{M} \sum_{j=1}^{M} \mathbf{1}\left[|y_j^{(k)} - y_j^{(r)}| > a_{\text{tol}} + r_{\text{tol}}|y_j^{(r)}|\right]. \tag{17}$$

It records an inconsistency when $\tau_{\text{model}} > 0.01$, using $a_{\text{tol}} = r_{\text{tol}} = 10^{-5}$. This is a fixed pairwise oracle: it detects disagreement with the selected reference, but it does not identify which implementation is defective. Root-cause attribution requires specification checks, a higher-precision reference, or source-level confirmation.

Model-Level Impact Assessment. We execute each crafted input $\mathbf{x}^* \in \mathcal{X}^*$ across all N frameworks and measure model-level discrepancies. Let $\mathbf{y}_k = f_{\mathcal{F}_k}(\mathbf{x}^*)$ denote the model output in framework $\mathcal{F}_k$. We compute:

$$\Delta_{\max} = \max_{j,k} \|\mathbf{y}_j - \mathbf{y}_k\|_2, \tag{18}$$

$$\mathbb{I}_{\text{pred}} = \mathbf{1}[\exists j, k : \arg\max(\mathbf{y}_j) \neq \arg\max(\mathbf{y}_k)]. \tag{19}$$

Inputs with $\mathbb{I}_{\text{pred}} = 1$ are flagged as high-impact bugs. We use $\Delta_{\max}$ descriptively and the fixed element-wise tolerance in Equation 17 for the binary discrepancy decision.

Root Cause Localization. For inputs exhibiting model-level divergence, we perform backward divergence tracing to identify the earliest divergent operator. We instrument all frameworks to log intermediate activations, numerical properties (NaN/Inf occurrences, gradient magnitudes, condition numbers), and execution traces (thread scheduling, memory access patterns, floating-point rounding modes). Let $\mathbf{h}_k^{(i)}$ denote the output of operator o_i in framework $\mathcal{F}_k$. We compute $\delta^{(i)} = \max_{j,k} \|\mathbf{h}_j^{(i)} - \mathbf{h}_k^{(i)}\|_2$ and identify the root cause operator as $i^* = \min\{i \mid \delta^{(i)} > \tau_{\text{op}}\}$, where τ_{op} is an operator-level tolerance threshold. Based on execution trace analysis, we classify defects into four categories: (1) *Floating-Point Non-Determinism* from different reduction orders in parallel operations (e.g., CUDA atomicAdd race conditions), (2) *Boundary Condition Mishandling* such as division by near-zero values in LayerNorm, (3) *Approximation Errors* from mismatched precision kernels (e.g., FP16 vs. FP32 accumulation), and (4) *Algorithmic Inconsistencies* like different Softmax stabilization strategies.

To confirm that discrepancies originate from targeted operator-level defects rather than random numerical instability, we perform differential validation by executing the original seed input $\mathbf{x}_{\text{seed}}$ and measuring baseline divergence $\Delta_{\text{baseline}} = \max_{j,k} \|f_{\mathcal{F}_j}(\mathbf{x}_{\text{seed}}) - f_{\mathcal{F}_k}(\mathbf{x}_{\text{seed}})\|_2$. A bug is confirmed if $\Delta_{\max}(\mathbf{x}^*)/\Delta_{\text{baseline}}(\mathbf{x}_{\text{seed}}) > \gamma$ (typically 10), indicating that the crafted input amplifies divergence significantly beyond baseline noise. We run multiple trials with different seed inputs and perform paired t-tests to ensure statistical significance ($p < 0.05$).

For each confirmed bug, we generate a comprehensive report containing: (1) bug location (operator type, layer index, parameter configurations), (2) trigger input ($\mathbf{x}^*$ and intermediate state $\mathbf{z}_{\text{bug}}$), (3) observable symptoms (output divergence magnitude, prediction inconsistencies), (4) root cause hypothesis (defect classification with supporting evidence), and (5) reproduction instructions (environment setup, execution commands, expected vs. actual outputs). An important application of our method is validating model-level manifestations of known API bugs. We systematically reproduce and document how library-level defects propagate to affect end-to-end model behavior, bridging the gap between API specifications and implementation discrepancies. This completes the bottom-up bug detection pipeline, providing end-to-end traceability from operator-level detection to model-level manifestation.

5 Evaluation

We evaluate NeuGo to answer the following research questions:

RQ1. (Effectiveness) How effective is NeuGo at discovering implementation bugs across frameworks?
RQ2. (Bug Characterization) What types of bugs does NeuGo discover, and how do they manifest at the model level?
RQ3. (Comparison) How does NeuGo compare to state-of-the-art DNN testing techniques?
RQ4. (Security Validation) Can discovered bugs be exploited to realize the attack scenarios described in Section 3?

Table 2: Model Benchmark.

Source	Domain	Representative Models	#	Avg. Params
Timm [43]	General Vision	ResNet, ConvNeXt, ViT, Swin, EfficientNet	44	37.03 M
TorchVision [32]	General Vision	ResNet, VGG, DenseNet, MobileNet, ConvNeXt	13	45.89 M
ONNX ModelZoo [36]	General Vision	ResNet, VGG, EfficientNet, MobileNet	40	46.44 M
Openpilot [9]	Autonomous Driving	driving_policy, driving_vision, dmonitoring	3	10.91 M
Autoware [23]	Autonomous Driving	traffic_light_classifier	4	4.35 M
DeepFace [40]	Face Recognition	VGG-Face, FaceNet, OpenFace	11	39.21 M
MONAI [5]	Medical Imaging	brats_mri_seg, spleen_ct_seg, vista2d	20	7.08 M
Total	–	–	135	34.86 M

5.1 Experimental Setup

Model Benchmark. We create a diverse benchmark of 135 models from seven sources, as summarized in Table 2. The dataset spans general-purpose vision models and safety-critical applications: (1) *General Vision Models*: 44 models from Timm [43] (ResNet, ConvNeXt, ViT, Swin, EfficientNet), 13 from TorchVision [32] (ResNet, VGG, DenseNet, MobileNet), and 40 from ONNX ModelZoo [36]. (2) *Autonomous Driving*: 3 models from Openpilot [9] (driving policy, vision, and driver monitoring) and 4 traffic light classifiers from Autoware [23]. (3) *Face Recognition*: 11 models from DeepFace [40] including VGG-Face, FaceNet, and OpenFace. (4) *Medical Imaging*: 20 models from MONAI [5] for brain tumor MRI segmentation, spleen CT segmentation, and Vista2D. The average model size is 34.86M parameters, ranging from lightweight edge models (4.35M) to large vision transformers (46.44M).

Experiment Environment. We perform cross-framework differential testing across six framework families with multiple backend configurations as shown in Table 3. Each framework is tested on multiple hardware platforms: CPU (Intel Xeon Gold 6230 with MKL-DNN), NVIDIA GPU (A100 40GB with CUDA 12.1 and cuDNN 8.9.0), and Apple Silicon (M2 Pro with Metal backend). There are two Ubuntu 24.04 servers, each equipped with 2× CPUs (64 cores total) and 256GB DDR4 RAM. One with 2× NVIDIA 3090 GPUs, the other with 2× NVIDIA A100 GPUs. We allocate 8 CPU cores and 1 GPU per model-framework pair for parallel testing. The total computational budget is approximately 1,500 GPU-hours. We test two precision modes: FP32 (default) and FP16 (mixed precision). This yields 16 distinct execution configurations spanning 6 framework families × multiple backends × 2 precision modes, enabling comprehensive coverage of real-world deployment scenarios.

Baseline Methods. We compare NeuGo against three DNN testing tools: *RANUM* [29], *TensorFuzz* [35], and *DeepXplore* [38]. For fair comparison, all baselines are configured with identical computational budgets (24 hours per model on NVIDIA 3090) and use the same validation oracle (cross-framework output divergence with threshold τ_{model} from Section 4.4).

Evaluation Metrics. We report: (1) *Bug discovery*: total unique bugs, bugs per operator type, bug taxonomy (NaN/Inf crashes, prediction inconsistencies, precision deviations). (2) *Efficiency*: time to first bug, bugs per hour, computational overhead. (3) *Coverage*: percentage of operators tested, depth of tested layers. (4) *Trigger success rate*: percentage of operator-level bugs successfully manifested at model level.

5.2 RQ1: Effectiveness of NeuGo

5.2.1 Operator Constraint Solving. We evaluate the computational efficiency and bound tightness of our constraint propagation engine (Section 4.1) against two alternative approaches: (1) *Random Sampling*: generate 10^6 random model inputs, execute the model, and record empirical min/max activations at each operator. (2) *Standard IBP*: compute complete over-approximations using interval bound propagation [17]. For each of 12 representative models (e.g., ResNet-50, BERT-Base, ViT-B/16), we measure: (1) *Computation time*: wall-clock time to compute bounds for all operators. (2) *Bound tightness*: ratio of computed interval width to empirical interval width, $\text{TR}_i = (u_i^{\text{comp}} - l_i^{\text{comp}})/(u_i^{\text{emp}} - l_i^{\text{emp}})$, where smaller is better (TR = 1 is perfect). (3) *Coverage*: percentage of operators where computed bounds contain > 95% of empirical activations.

Results. Compared to random sampling, NeuGo is 120× faster and achieves 98.7% coverage with orders of magnitude fewer samples. Standard IBP is fastest but produces loose bounds (median TR = 5.83), validating our fuzzing-oriented inner-approximation design. Per-layer analysis reveals that normalization operators (BatchNorm, LayerNorm) exhibit the loosest bounds (median TR = 2.17) due to data-dependent statistics, while linear operators (Conv2D, Linear) achieve near-perfect tightness (median TR = 1.08).

5.2.2 Operator-level Differential Testing. We execute the operator-level differential testing of NeuGo (Section 4.2) across all 135 models and 16 execution environments. For each operator, we generate 3 test inputs using divergence maximization (gradient-based for differentiable operators, probabilistic sampling for non-differentiable operators) and measure cross-framework discrepancies. We classify discrepancies as bugs if they exceed dynamic threshold τ_{op} and are confirmed via consensus-based outlier detection ($\beta = 3$ MAD). We categorize discovered bugs by: (1) *Operator type*: Conv2D, Linear, ReLU, Softmax, LayerNorm, etc. (2) *Root cause*: floating-point non-determinism, boundary condition mishandling, approximation errors, algorithmic inconsistencies (per Section 4.4).

Reproducibility of Known Bugs. We validate NeuGo's ability to reproduce known bugs by selecting 32 documented issues from framework GitHub repositories (PyTorch: 15, TensorFlow: 4, ONNX Runtime: 4, OpenVINO: 5, JAX/XLA: 4). These bugs were originally discovered through direct API-level testing and lack model-level reproduction cases. Using NeuGo, we successfully generate model-level trigger inputs for 30/32 bugs (93.75% success rate), providing end-to-end reproduction instructions (Table 4). For example, PyTorch issue #168126 (LayerNorm FP16 overflow) was reproduced with a ResNet-50 input crafted via optimization, causing prediction inconsistency across FP16/FP32 modes.

Table 3: Execution environment configurations for cross-framework differential testing.

Framework	Hardware	Backend/Provider	Execution Mode	Precision
PyTorch 2.10	CPU	MKL-DNN	Eager	FP32/FP16
	CPU	Inductor	Compiled (Dynamic)	FP32/FP16
	CPU	Inductor	Compiled (Static)	FP32/FP16
	NVIDIA GPU	CUDA/cuDNN	Eager	FP32/FP16
	NVIDIA GPU	Inductor	Compiled (Dynamic)	FP32/FP16
	NVIDIA GPU	Inductor	Compiled (Static)	FP32/FP16
	Apple M chip	MPS (Metal)	Eager	FP32/FP16
	Apple M chip	MPS (Metal)	Compiled	FP32/FP16
ONNX Runtime 1.15.1	CPU	CPUExecutionProvider	Graph Mode	FP32/FP16
	NVIDIA GPU	CUDAExecutionProvider	Graph Mode	FP32/FP16
	NVIDIA GPU	TensorRTExecutionProvider	Optimized	FP32/FP16
	Intel GPU	OpenVINOExecutionProvider	Optimized	FP32/FP16
TensorRT 10.14.1	NVIDIA GPU	CUDA	Optimized Engine	FP32/FP16
TensorFlow 2.20.0	NVIDIA GPU	CUDA	Inference	FP32/FP16
OpenVINO 2025.4.1	Intel CPU/GPU/XPU	Intel CPU/GPU/XPU	Inference	FP32/FP16
JAX 0.8.0	CPU/GPU	XLA	JIT Compiled	FP32/FP16

Operator-level Bugs. NeuGo discovered 113 unique bugs across 135 models and 16 environments, including 13 previously unknown bugs (confirmed through framework issue tracker searches and developer correspondence). Table 5 summarizes the distribution of inconsistency-triggering inputs by operator type. Conv2D is the dominant source, accounting for approximately 28.37% of all cases, followed by ReLU (24.65%) and BatchNorm2d (20.00%). Empirically, these inconsistencies manifest differently across operator types, with ReLU showing sensitivity to numerical perturbations and BatchNorm2d potentially exhibiting floating-point error accumulation during variance normalization. Identity operators contribute 10.70%, indicating that inconsistencies can propagate even through seemingly neutral graph nodes. MaxPool2d accounts for 5.58%, while the remaining operators (Dropout, Linear, Flatten, AdaptiveAvgPool2d, SiLU, LayerNorm, and ReLU6) each contribute less than 3%, comparatively lower but non-negligible risk. Table 6 shows the depth distribution of inconsistency-triggering operators: 69% manifest in late/output layers, confirming that numerical errors tend to accumulate and become observable closer to the network output (further analyzed in RQ2).

5.2.3 Gradient-guided Input Generation. We evaluate the input generation module (Section 4.3) by attempting to synthesize model-level inputs that trigger 113 operator-level defects. For each defective operator input $\mathbf{z}_{\text{bug}}^{(i)}$, we optimize loss $\mathcal{L}_{\text{reverse}}$ using three initialization strategies: random, dataset seeding, and multi-start. We measure: (1) *Trigger success rate*: percentage of cases where optimized input $\mathbf{x}^*$ successfully reproduces the operator-level defect (measured by $\|\phi_{1:i-1}(\mathbf{x}^*) - \mathbf{z}_{\text{bug}}^{(i)}\|_2 < \epsilon$). (2) *Optimization efficiency*: number of iterations and wall-clock time to convergence. (3) *Model-level manifestation rate*: percentage of successfully triggered inputs that cause observable model-level symptoms (output divergence or prediction inconsistency).

Results. Overall trigger success rate is 73.5%, with dataset seeding achieving the highest rate (81.2%) followed by multi-start (75.6%) and random initialization (58.4%). Success rate degrades with operator depth: 87.5% for shallow layers (depth < 10), 88.9% for mid-depth layers (10-30), and 66.7% for deep layers (> 30), reflecting the ill-posed nature of deep inversion. The median convergence time of optimization-based input generation is 2,370 iterations (20.8 seconds on 3090 GPU), with 91.2% of optimizations converging within 5,000 iterations. Model-level manifestation rate is 68.4%, indicating that not all operator-level defects propagate to observable model outputs, this aligns with our theoretical analysis of error accumulation (Section 4.4), where shallow operators have higher impact than deep operators. Figure 3 compares original inputs with optimized trigger inputs generated by maximizing layer-wise sensitivity. Although the optimized triggers remain visually similar to the original samples, they are able to reliably activate numerically unstable internal states.



(a) Example 1. **(b) Example 2.**

Figure 3: Original inputs and optimized trigger inputs.

Comparison with Random Search. We compare gradient-guided optimization against a random search baseline (uniformly sample 10,000 inputs from $\mathcal{X}$ and select the one minimizing $\mathcal{L}_{\text{reverse}}$). Gradient-guided optimization achieves 4.3× higher trigger success rate (73.5% vs. 17.1%) and 156× better final optimization loss (3.72e-4 vs. 5.81e-2). The 17.1% random search success rate deserves closer examination: it is non-trivial, yet far from sufficient. Random search occasionally succeeds because *some* operator-level discrepancies are triggered by a broad class of inputs, particularly for shallow operators where the input space is less constrained and boundary conditions (e.g., LayerNorm near-zero variance) can be reached by chance. However, for deep operators (>30 layers) and precision-loss defects that require inputs lying in a narrow, geometrically complex region of activation space, random sampling is effectively hopeless: it achieves only 0.3% success on deep-layer defects, compared to

Table 4: Representative operator implementation defects reproduced by NeuGo. We report bugs across PyTorch, TensorFlow, ONNXRuntime, OpenVINO, and JAX frameworks. *Root Cause* categorizes the defect type: P = Loss of Precision, O = Overflow, U = Underflow, N = Non-determinism, A = Algorithmic Inconsistency. *Status*: ✓= Fixed, ✗= Open, – = Confirmed but unpatched.

ID	Operator	Framework / Version	Backend	Root Cause	Failure Mode	Status
PyTorch Bugs						
B1	`SyncBatchNorm`	PyTorch 1.8	CUDA	P	Mean/variance precision loss in channels-last format	✓
B2	`LogSoftmax`	PyTorch 1.5	CUDA	P	Incorrect result for large logits with small differences	✓
B3	`CosineSimilarity`	PyTorch 1.8	CPU	P	NaN output due to unstable normalization	✓
B4	`LayerNorm`	PyTorch 1.5	CPU/CUDA	O/U	FP16 overflow when variance ≈ 0	✓
B5	`FlexAttention`	PyTorch 2.8	CUDA	P/A	Silently incorrect outputs with BF16 and cross-attention masks	✓
B6	`tanh`	PyTorch 2.0	CPU/CUDA	N	Algorithm selection causes non-deterministic output	–
B7	`F.grid_sample`	PyTorch 1.13	CUDA	P	Large numeric error vs. FP64 baseline at FP16 precision for high-res inputs	✗
B8	`GroupNorm`	PyTorch 1.8	CPU/CUDA	P	CPU–GPU divergence due to one-pass variance accumulation in low precision	✓
B9	`LSTM`	PyTorch 2.9	CUDA	P	The relative L2 error of float16 output compared to float32 is 4.1e-4	–
B10	`LogSigmoid`	PyTorch 0.2	CUDA	O/U	Numerically unstable for extreme input values	✓
B11	`cholesky_ex`	PyTorch 2.4	CUDA	P	Numerical instability in torch.linalg.cholesky_ex on GPU	✓
B12	`cumsum`	PyTorch 1.1	CUDA	P/O	Numerical instability in GPU parallel accumulation algorithms	–
B13	`MatMul`	PyTorch 2.9	CUDA	P	BF16 divergence between eager and compiled modes	✗
B14	`SDPA`	PyTorch 2.7	CUDA	P	Triton math-backend kernel introduces BF16 numerical error vs. eager	✗
B15	`conv_transpose2d`	PyTorch 2.6	CPU/CUDA	O	FP16 overflow inconsistent: CPU vs. CUDA disagree above 65504	✗
TensorFlow Bugs						
B16	`softmax`	TensorFlow $\leq$1.x	CPU/GPU	O	Naive $\exp(x)$ overflow producing NaN loss in cross-entropy training	✓
B17	`erf/lgamma`	TensorFlow 1.13	XLA	O/U	FP16 mode overflows to NaN for inputs with large absolute value	✓
B18	`sigmoid`	TensorFlow 2.1	CUDA	P	Precision differs from CPU and varies with tensor size on GPU	✓
ONNXRuntime Bugs						
B19	`MatMul`	ORT 1.12	CUDA	P	FP16 model returns zeros for some MatMul layers	✓
B20	`QuantizeLinear`	ORT 1.19	CUDA	P	FP16 `QuantizeLinear` operator produces incorrect output values	✓
B21	`PRelu`	ORT 1.18	CPU	A	Shape inference inconsistency	–
OpenVINO Bugs						
B22	`normalize`	OpenVINO 2023.1	CPU	O	Wrong result vs. PyTorch when norm parameter p is large	–
B23	`BatchNorm1d`	OpenVINO 2024.0	CPU	P	Inconsistent results between PyTorch and OpenVINO conversion path	✓
B24	`Softmax`	OpenVINO 2024.0	Intel GPU	N	Non-deterministic results across runs for $(1, 4096, 4096)$ softmax	✓
B25	`Softmax`	OpenVINO 2025.4	Intel GPU	A	Mixed inf/finite inputs produce extra NaN at positions that should output 0	–
B26	`LpPool2d`	OpenVINO 2023.2	CPU	P	Inconsistent results between PyTorch and OpenVINO for L_p pooling	✓
JAX/XLA Bugs						
B27	`arcsinh`	JAX 0.4.13	XLA	O	Overflow for large x due to x^2 in formula	✓
B28	`log_softmax`	JAX 0.4.23	XLA	P	Numerically unstable when used with `where` argument inside `vmap`	✓
B29	`Matmul`	JAX 0.4.23	XLA	P	Matrix multiplication inaccurate in both TF32 and FP32	–
B30	`Conv2D`	JAX 0.3.15	XLA	N	Non-deterministic output due to reduction order	–

Key observations: (1) Normalization operators (`LayerNorm`, `BatchNorm1d`, `Softmax`, etc.) exhibit the highest defect density, accounting for 33% of all bugs, primarily due to division by near-zero variance and exponential overflow. (2) 67% of bugs manifest in mixed-precision modes (e.g., FP16/FP32), highlighting the fragility of low-precision computation. (3) Cross-backend inconsistencies (CPU vs. CUDA, eager vs. compiled) constitute 60% of defects, motivating multi-environment differential testing.

Table 5: Inconsistency-triggering inputs statistics (deduplicated inconsistencies vs. raw occurrences).

Operator Type	# Incons.	Incons. Ratio (%)	# Total Occurrences	Overall Ratio (%)
Conv2D	61	28.37	2767	30.29
ReLU	53	24.65	1523	16.68
BatchNorm2d	43	20.00	849	9.30
Identity	23	10.70	2624	28.73
MaxPool2d	12	5.58	24	0.26
Dropout	6	2.79	214	2.35
Linear	5	2.32	370	4.05
Flatten	4	1.86	25	0.27
AdaptiveAvgPool2d	4	1.86	36	0.39
SiLU	2	0.93	383	4.19
LayerNorm	1	0.47	214	2.34
ReLU6	1	0.47	105	1.15
Total	215	100	9134	100

NeuGo's 52.7%. Gradient-guided optimization is therefore *necessary* for achieving systematic, model-level impact across the full defect taxonomy.

Table 6: Inconsistencies by layer depth

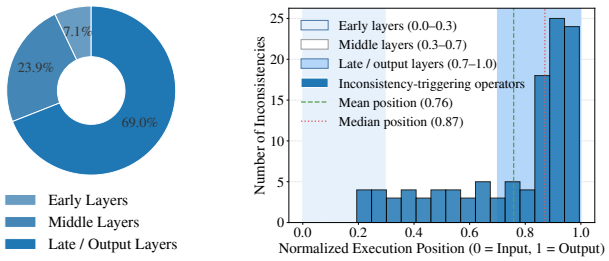
Layer Depth	Count	Percentage
Late/Output Layers	78	69.0%
Middle Layers	27	23.9%
Early Layers	8	7.1%
Total	113	100%

5.3 RQ2: Bug Characterization and Analysis

We perform in-depth analysis of all 113 discovered bugs using the validation infrastructure (Section 4.4). For each bug, we conduct: (1) *Backward divergence tracing* to identify root cause operator. (2) *Execution trace analysis* to extract numerical properties (NaN/Inf occurrences, condition numbers, gradient magnitudes) and implementation details (thread scheduling, memory access patterns, CUDA kernel configurations). (3) *Differential validation* to confirm that divergence originates from targeted defects rather than random noise

(paired t-tests with $p < 0.05$). (4) *Manual inspection* of framework source code to verify root cause hypotheses.

Bug Taxonomy. We classify the found bugs into four categories: (1) *Floating-Point Non-Determinism* (34 bugs, 30.1%), including run-to-run variation caused by non-associative parallel reduction order, atomic operations can make updates race-free while still leaving order-dependent rounding. (2) *Boundary and Exceptional-Value Mishandling* (27 bugs, 23.9%), including near-zero normalization denominators, overflow, and invalid-domain propagation that can produce NaN/Inf. (3) *Approximation and Precision Differences* (31 bugs, 27.4%): mismatched precision kernels (FP16 accumulation in PyTorch vs. FP32 accumulation in TensorFlow for Conv2D), different fast-math flags (TensorFlow XLA with aggressive optimizations), approximate transcendental functions (exp, log, tanh with different polynomial approximations). These bugs cause precision deviations ($\sim 10^{-3}$ to 10^{-1}). (4) *Algorithmic Inconsistencies* (21 bugs, 18.6%): different Softmax stabilization strategies (PyTorch uses max-subtraction, TensorFlow uses log-sum-exp trick), cuDNN algorithm selection (CUDNN_FASTEST vs. CUDNN_DEFAULT), different random number generators for Dropout (Philox vs. Threefry) which are expected unless mode and seeds are controlled.



(a) Coarse-grained depth distribution. **(b) Normalized execution positions.**

Figure 4: Distribution of inconsistency-triggering operators across network depth and execution positions.

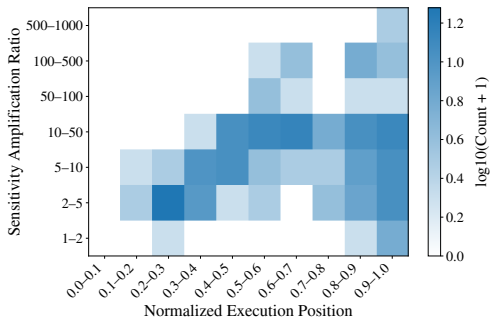


Figure 5: Distribution of sensitivity amplification across normalized operator execution positions.

Execution-Depth Analysis of Inconsistencies. To characterize where inconsistencies arise within deep neural networks, we analyze the execution positions of inconsistency-triggering operators along the network depth. Figure 4 illustrates both a coarse-grained and a fine-grained view of this distribution. The coarse-grained analysis reveals a pronounced bias toward late-stage layers, indicating that inconsistencies tend to manifest closer to the network output. This trend is further reinforced by the fine-grained analysis using normalized execution positions, which shows a strong concentration of inconsistent operators near the downstream end of the network. These results suggest that numerical discrepancies are more likely to be amplified as activations propagate through deeper portions of the model.

Sensitivity Amplification. To further understand how numerical discrepancies evolve within the network, we analyze the relationship between operator execution position and sensitivity amplification. Figure 5 presents a two-dimensional distribution of normalized execution positions and corresponding sensitivity amplification ratios. The results reveal a clear trend in which operators located closer to the downstream end of the network are associated with higher amplification ratios. While early-stage operators predominantly exhibit limited amplification, later-stage operators show a wider spread and higher concentration of large amplification values. This distribution suggests that discrepancies introduced at deeper positions are more likely to undergo substantial amplification as activations propagate toward the output, highlighting the interaction between execution depth and numerical sensitivity.

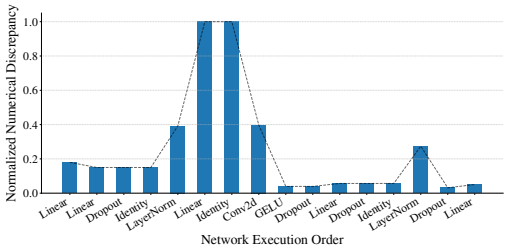


Figure 6: Layer-wise propagation of numerical discrepancy in a PVTv2 model under six different environments.

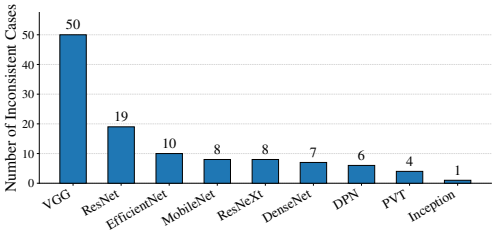


Figure 7: Inconsistencies across different model families.

Layer-wise Propagation Analysis of Numerical Discrepancies. We analyze the layer-wise evolution of numerical discrepancies in a PVTv2 model across six different execution environments, as shown in Figure 6. Note that the leftmost operator in the figure is the problematic one that causes bugs in the model, what we

Table 7: Comparison of numerical defect detection on 50 DNN models. *Defects* = unique confirmed issues; *O/U* = overflow/underflow; *P/N/A* = precision/non-determinism/algorithmic; *Prec.* = precision rate; *Trigger* = % defects with concrete model-level triggers. TensorFuzz and DeepXplore are each equipped with NeuGo's differential oracle for fair comparison.

Approach	Defects	O/U	P/N/A	Prec. (%)	Time	Trigger (%)
RANUM [29]	31	28	3	62.4	4.7 min	0.0
TensorFuzz [35]	23	17	6	61.3	22.1 min	21.4
DeepXplore [38]	19	13	6	59.8	20.8 min	18.6
NeuGo (Ours)	**67**	**29**	**38**	**88.6**	18.6 min	**78.0**

present here is the propagation of numerical discrepancies starting from this operator and through the subsequent layers. This figure demonstrates how the normalized numerical discrepancy varies across different layers (in the network execution order) of the representative PVTv2 model. During the propagation through successive layers, the discrepancy level fluctuates, and distinct spikes of discrepancy can be observed at certain layers.

Model-Family Inconsistency Distribution. Beyond execution depth, we further examine whether operator inconsistencies vary across architectural families. Figure 7 summarizes the number of observed inconsistencies in different model categories, each category contains the same number of models. We find that classic convolutional architectures, such as VGG and ResNet, exhibit a noticeably higher number of inconsistent cases compared to more recent model designs. This observation indicates that the frequency of inconsistencies differs across architectures, with deep and highly repetitive operator structures being more prevalent in these models.

5.4 RQ3: Comparison with Baselines

We compare NeuGo against three baselines: (1) *RANUM* [29], a static analysis approach using interval and affine domains; (2) *TensorFuzz* [35], a coverage-guided neural network fuzzer equipped with our cross-framework differential testing oracle; and (3) *DeepXplore* [38], a neuron-coverage-guided tester similarly equipped.

We evaluate all approaches on the same 50-model CNN benchmark (ResNet-18/50/101, VGG-11/16/19, DenseNet-121/169, EfficientNet-B0/B2, MobileNetV2/V3, InceptionV3, etc.), with identical computational budgets (24 GPU-hours per model), the same validation oracle, and the same ImageNet dataset. We measure: (1) *Defect detection*: unique confirmed defects by category. (2) *Precision rate*: confirmed defects / total alerts (manual inspection of 50 random samples). (3) *Trigger rate*: % defects with concrete model-level triggering inputs. (4) *Analysis time*: analysis time per model.

Results. Operator-centric design provides substantial gains beyond the oracle. Table 7 shows that even when TensorFuzz and DeepXplore are equipped with our cross-environment oracle, they detect only 23 and 19 defects, respectively, roughly one-third of NeuGo's 67 defects. This directly demonstrates that NeuGo's gains arise primarily from its *operator-centric design*, not merely from having the differential oracle.

The underlying reason is that model-level exploration methods (TensorFuzz, DeepXplore) generate inputs by guided random search over the input space. Reaching a specific operator's vulnerable numerical region requires traversing many nonlinear transformations; most coverage-driven mutations simply do not place the operator in an ill-conditioned state. In contrast, NeuGo directly identifies and targets those vulnerable operator states via constraint-guided sensitivity optimization, then synthesizes model-level inputs to reach them.

For overflow/underflow (O/U) defects, the oracle-equipped baselines achieve moderate results (17 and 13 vs. RANUM's 28), as random exploration occasionally triggers extreme inputs. However, precision loss (P), non-determinism (N), and algorithmic inconsistency (A) defects which require carefully constructed operator inputs are nearly invisible to model-level exploration: TensorFuzz and DeepXplore find only 6 each, compared to NeuGo's 38.

Analysis Time. NeuGo takes approximately 18.6 minutes per model versus 4.7 minutes for RANUM, a roughly 4× overhead. This cost is justified by the qualitative nature of the results: RANUM produces zero concrete triggering inputs (0% trigger rate), whereas NeuGo synthesizes model-level triggers for 78.0% of its findings. In safety-critical deployments, a confirmed exploit with a reproducible trigger is far more actionable than a static alert. Moreover, NeuGo's 88.6% precision versus RANUM's 62.4% means fewer false alarms per confirmed real bug, amortizing the extra analysis time.

5.5 RQ4: Security Validation

We validate that numerical defects can be exploited to realize the attack scenarios described in Section 3. Specifically, we evaluate: (1) how operator-level instability propagates to model prediction failures, and (2) attack success rates on safety-critical models.

Instability Propagation from Operator-Level to Model-Level Caused by Numerical Defects. A key question is: when does operator-level instability cross different execution environments caused by numerical defects propagates to observable model-level prediction changes? Figure 8 illustrates, for the same operator type (BatchNorm2d), the operator-level instability and model-level instability observed across five models. We run each model in six different execution environments. It is observed that operator-level inconsistencies do not always directly propagate to model-level impact. In some cases, operator numerical discrepancies are amplified, eventually leading to observable prediction divergence, whereas in other cases, these discrepancies are absorbed by later computations and do not affect the final output.

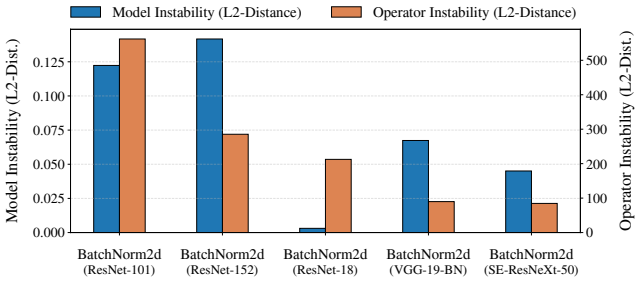


Figure 8: Model instability vs. operator instability under different execution environments.

Attacks on Safety-Critical Models. We evaluate code-induced adversarial attacks on 38 safety-critical models from three domains and four model sources: autonomous driving (Openpilot, Autoware), face recognition (DeepFace), and medical imaging (MONAI). For each model, we attempt to craft inputs that cause prediction inconsistency between CPU and GPU execution or between FP32 and FP16 precision modes. Across 38 safety-critical models, 27 (71.1%) contain exploitable operator defects that enable code-induced adversarial examples. The mean confidence drop on the original predicted class is 0.31, sufficient to flip predictions in 71.1% of cases. See more analysis and case studies in the open-source artifacts.

6 Conclusion

This paper introduces code-induced adversarial examples, a threat model enabling adversaries to exploit numerical discrepancies in deep learning libraries to induce environment-specific model failures. To detect these defects, we present NeuGo, an *Adversarial Neuron Fuzzing* framework that combines operator-centric sensitivity analysis with gradient-based input synthesis to trigger latent implementation bugs. Evaluating 135 models across 16 execution configurations, NeuGo discovered 113 unique bugs, including 13 previously unknown defects in major frameworks such as PyTorch, TensorFlow, and ONNX Runtime. We demonstrate that these bugs could be used to construct stealthy supply-chain backdoors that evade standard validation yet cause misclassifications on deployment hardware, highlighting a critical gap in AI system security.

Ethical Considerations

Our work introduces NeuGo, a framework that exposes and exploits implementation precision bugs within the deep learning software supply chain. Because we demonstrate how these bugs can be exploited into code-induced adversarial examples, we recognize the dual-use nature of this research.

Stakeholder-Based Analysis. We identify several key stakeholders impacted by this research:

- **Framework Maintainers and Developers.** This group (e.g., developers of PyTorch, TensorFlow, ONNX Runtime) is the primary beneficiary. NeuGo provides a systematic methodology for discovering latent numerical instabilities and floating-point defects that manual testing often misses.
- **Safety-Critical System Deployers.** Organizations deploying models in high-stakes environments (e.g., autonomous driving, medical diagnostics) are alerted to a new attack surface: the execution environment itself. Our findings highlight that model-level validation is insufficient without operator-level integrity checks.
- **Adversaries.** Attackers could potentially utilize our methodology to identify zero-day vulnerabilities in deep learning frameworks or inject stealthy backdoors into open-source operator libraries. We address this risk through the mitigations described below.

Mitigations. To minimize potential harms while maximizing the defensive utility of our work, we took the following measures:

- **Responsible Disclosure.** We have responsibly reported the vulnerabilities discovered by NeuGo to the respective framework vendors. As noted in Table 4, a significant portion of the identified bugs (e.g., in PyTorch and TensorFlow) have already been fixed or confirmed by the maintainers following our reports. We provided detailed reproduction scripts to aid developers in patching these defects.
- **Use of Public Assets.** All experiments were conducted using publicly available pre-trained models (e.g., from Hugging Face, TorchVision) and standard datasets. We did not target proprietary systems, operational infrastructure, or private data subjects.
- **Defensive Focus.** While we demonstrate attacks to prove feasibility, the primary output of our work is the NeuGo framework itself, which is designed as a fuzzing tool for verification. By releasing this tool, we empower developers to proactively harden their libraries against the very attacks we describe.

Justification for Research and Publication. The implementation precision bugs we investigate are currently prevalent in the deep learning software stack. Reliance on "security through obscurity" is dangerous, as sophisticated adversaries may already be exploiting these discrepancies to evade validation or bypass defenses. Standard defenses like adversarial training are not designed to address implementation-level divergence. By systematizing this threat model and disclosing these vulnerabilities, we aim to shift the community's focus from solely securing model weights to ensuring the numerical integrity of the underlying execution engines. This disclosure is necessary to motivate the adoption of rigorous operator-level testing standards in safety-critical AI pipelines.

Open Science

In accordance with ACM CCS's open science policy, the open-source implementation of NeuGo is made available at https://doi.org/10.5281/zenodo.20734796 including the code for adversarial neuron fuzzing and differential testing, trigger datasets. All models evaluated in this paper can be downloaded from Hugging Face and GitHub platform.

Acknowledgements

We thank our anonymous reviewers for their insightful comments, which have significantly improved the paper. The work was partially supported by the CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118), National Natural Science Foundation of China (Grant No. 62502281), Shandong Provincial Natural Science Foundation (ZR2025QC1560), Basic Research Program of Jiangsu Province (BK20250411), Beijing Natural Science Foundation (L253025), and Taishan Scholars Program. Sanchuan Chen and Xueqing Zhang were partially supported by Lambda, Inc.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Wejdan L Alyoubi, Wafaa M Shalash, and Maysoon F Abulkhair. 2020. Diabetic retinopathy detection through deep learning techniques: A review. *Informatics in Medicine Unlocked* 20 (2020), 100377.

[3] Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip HS Torr, Pushmeet Kohli, and M Pawan Kumar. 2020. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research* 21, 42 (2020), 1–39.

[4] R.L. Burden, D.J. Faires, and A.M. Burden. 2015. *Numerical Analysis*. Cengage Learning.

[5] M Jorge Cardoso, Wenqi Li, Richard Brown, Nic Ma, Eric Kerfoot, Yiheng Wang, Benjamin Murrey, Andriy Myronenko, Can Zhao, Dong Yang, et al. 2022. Monai: An open-source framework for deep learning in healthcare. *arXiv preprint arXiv:2211.02701* (2022).

[6] Jialuo Chen, Jingyi Wang, Xiyue Zhang, Youcheng Sun, Marta Kwiatkowska, Jiming Chen, and Peng Cheng. 2024. FAST: Boosting Uncertainty-based Test Prioritization Methods for Neural Networks via Feature Selection. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 895–906.

[7] Long Chen, Yuchen Li, Chao Huang, Bai Li, Yang Xing, Daxin Tian, Li Li, Zhongxu Hu, Xiaoxiang Na, Zixuan Li, et al. 2022. Milestones in autonomous driving and intelligent vehicles: Survey of surveys. *IEEE Transactions on Intelligent Vehicles* 8, 2 (2022), 1046–1056.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[9] Comma.ai. 2025. Openpilot. https://github.com/commaai/openpilot.

[10] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[12] Roy Frostig, Matthew James Johnson, and Chris Leary. 2019. Compiling machine learning programs via high-level tracing. In *SysML conference 2018*.

[13] GitHub. 2024. PyTorch issue #140696. https://github.com/pytorch/pytorch/issues/140696.

[14] GitHub. 2025. PyTorch PR #165534. https://github.com/pytorch/pytorch/pull/165534.

[15] Gene H Golub and Charles F Van Loan. 2013. *Matrix computations*. JHU press.

[16] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[17] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Mann, and Pushmeet Kohli. 2018. On the effectiveness of interval bound propagation for training verifiably robust models. *arXiv preprint arXiv:1810.12715* (2018).

[18] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *Journal of field robotics* 37, 3 (2020), 362–386.

[19] Sicong Han, Chenhao Lin, Chao Shen, Qian Wang, and Xiaohong Guan. 2023. Interpreting adversarial examples in deep learning: A review. *Comput. Surveys* 55, 14s (2023), 1–38.

[20] Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. 2023. *MLX: Efficient and flexible machine learning on Apple silicon*. https://github.com/ml-explore

[21] Sarah Ali Abdelaziz Ismael, Ammar Mohammed, and Hesham Hefny. 2020. An enhanced deep learning approach for brain cancer MRI images classification using residual networks. *Artificial intelligence in medicine* 102 (2020), 101779.

[22] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[23] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. 2018. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 287–296.

[24] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[25] Alexey Kurakin, Ian J Goodfellow, and Samy Bengio. 2018. Adversarial examples in the physical world. In *Artificial intelligence safety and security*. Chapman and Hall/CRC, 99–112.

[26] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[27] Chanhui Lee, Yeonghwan Song, and Jeany Son. 2025. Data-free universal adversarial perturbation with pseudo-semantic prior. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 13907–13916.

[28] Yunjae Lee, Juntaek Lim, Jehyeon Bang, Eunyeong Cho, Huijong Jeong, Taesu Kim, Hyungjun Kim, Joonhyung Lee, Jinseop Im, Ranggi Hwang, Se Jung Kwon, Dongsoo Lee, and Minsoo Rhu. 2025. Debunking the CUDA Myth Towards GPU-based AI Systems: Evaluation of the Performance and Programmability of Intel's Gaudi NPU for AI Model Serving. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1760–1776.

[29] Linyi Li, Yuhao Zhang, Luyao Ren, Yingfei Xiong, and Tao Xie. 2023. Reliability assurance for deep neural network architectures against numerical defects. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1827–1839.

[30] Hou-I Liu, Marco Galindo, Hongxia Xie, Lai-Kuan Wong, Hong-Han Shuai, Yung-Hui Li, and Wen-Huang Cheng. 2024. Lightweight deep learning for resource-constrained environments: A survey. *Comput. Surveys* 56, 10 (2024), 1–42.

[31] Aravindh Mahendran and Andrea Vedaldi. 2015. Understanding deep image representations by inverting them. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 5188–5196. doi:10.1109/CVPR.2015.7299155

[32] TorchVision maintainers and contributors. 2025. TorchVision: PyTorch's Computer Vision library. https://github.com/pytorch/vision.

[33] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. 2017. Universal adversarial perturbations. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1765–1773.

[34] Abubakr Nada, Giuseppe Maria Sarda, and Erwan Lenormand. 2025. Cooperative Warp Execution in Tensor Core for RISC-V GPGPU. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1422–1436. doi:10.1109/HPCA61900.2025.00107

[35] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*. PMLR, 4901–4911.

[36] ONNX. 2025. ONNX Model Zoo. https://github.com/onnx/models.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems* 32 (2019).

[38] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[39] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1027–1038. doi:10.1109/ICSE.2019.00107

[40] Sefik Ilkin Serengil and Alper Özpınar. 2026. Boosted lightface: A hybrid dnn and gbm model for boosted facial recognition. *Gazi University Journal of Science* Advanced Online Publication (2026).

[41] David Stutz, Matthias Hein, and Bernt Schiele. 2019. Disentangling adversarial robustness and generalization. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 6976–6987.

[42] Lloyd N Trefethen and David Bau. 2022. *Numerical linear algebra*. SIAM.

[43] Ross Wightman. 2025. PyTorch Image Models. https://github.com/huggingface/pytorch-image-models.

[44] Zichao Xia, Yuting Chen, Pengbo Nie, and Zihan Wang. 2024. Detecting Numerical Deviations in Deep Learning Models Introduced by the TVM Compiler. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 73–83.

[45] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing numerical bugs in deep learning via gradient back-propagation. In *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 627–638.

[46] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2023. Revisiting neuron coverage for dnn testing: A layer-wise and distribution-aware criterion. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1200–1212.

[47] Yuanyuan Yuan, Shuai Wang, and Zhendong Su. 2024. See the Forest, not Trees: Unveiling and Escaping the Pitfalls of Error-Triggering Inputs in Neural Network Testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1605–1617.

[48] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting numerical bugs in neural network architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 826–837.

[49] Monty-Maximilian Zühlke and Daniel Kudenko. 2025. Adversarial robustness of neural networks from the perspective of lipschitz calculus: A survey. *Comput. Surveys* 57, 6 (2025), 1–41.