FAULT INTERNALIZATION: WHEN TRAINING OPTIMIZATION ABSORBS INJECTED HARDWARE FAULTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Outsourced AI model training and fine-tuning increasingly rely on third-party platforms, where trusted execution environments (TEEs) are widely adopted to isolate sensitive model weights and training data from the untrusted privileged host. However, hardware fault injection through bit-flip attacks (BFAs) remains capable of perturbing protected computation. Existing BFAs either require white-box access to model weights or destructively corrupt low-level execution, making them unsuitable for practical training-stage attacks. We uncover *fault internalization*, a training-specific phenomenon in which optimization adapts to altered computation by encoding compensation into the learned weights. An untrusted host can exploit this behavior to keep training and validation normal when performing BFAs, thereby concealing the ongoing attack. Once the model is transferred to a clean platform, the original computation is restored while the learned compensation remains encoded in the weights. The compensation then no longer offsets the altered computation and instead drives the model toward the adversarial behavior. We characterize the semantic- and implementation-level conditions that enable fault internalization and develop a systematic procedure for mapping suitable computation alterations to target bits in underlying computation libraries. We evaluate fault internalization on discriminative and generative models under training-from-scratch, full-weight fine-tuning, and LoRA fine-tuning. Across these settings, attacked execution remains close to benign behavior, whereas restoring the original computation on a benign platform exposes diverse model-level failures.

1 INTRODUCTION

Training and fine-tuning increasingly rely on third-party computing platforms to access the substantial computational resources required by modern AI models. However, entrusting model execution to an external host exposes proprietary models and training data to privileged system software, which may inspect or arbitrarily tamper with the outsourced workload. Trusted execution environments (TEEs) have therefore emerged as a practical protection mechanism, offering hardware-enforced isolation with relatively low overhead and broad compatibility with existing AI software stacks Kunkel et al. (2019); Zhang et al. (2021). While TEEs restrict privileged software from directly accessing protected models and data, hardware fault injection can still perturb the computation executed within the protected environment Carru (2017); Murdock et al. (2020); Chen et al. (2021b); Qiu et al. (2020). Consequently, bit-flip attacks (BFAs) Rakin et al. (2019), a representative form of such fault injection, remain a practical threat to protected AI training on untrusted platforms.

Existing BFAs against AI systems mainly follow two directions. Weight-level attacks identify and flip critical bits in model weights, typically requiring white-box access to the model and the ability to compute gradients over its weights to localize vulnerable bits Rakin et al. (2019; 2021); Bai et al. (2021); Xu et al. (2026). These requirements are incompatible with TEE-protected training, where the untrusted host is prevented from accessing such protected model state. Library- or code-level BFAs provide a more realistic alternative for a privileged host, as they manipulate the underlying execution stack without directly accessing protected model states; the corresponding code and implementation details are typically available to the host. However, existing attacks in this category have largely targeted inference by destructively perturbing execution, for example, by disrupting control/data flow Li et al. (2024) or corrupting program instructions Chen et al. (2025a). Although such perturbations can severely impair model execution while keeping the program running, apply-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

ing the same destructive paradigm to training will break the training process, violating the stealth requirement of the attack.

A training-stage attack thus requires a different strategy: instead of destructively *corrupting* execution, computation semantics should be *altered* while making the training process appear normal. With deliberate modifications such as replacing the $\max(\cdot)$ operation in ReLU with $\min(\cdot)$, we find that optimization can absorb the resulting computational changes into the learned weights. Specifically, as training proceeds, the weights adapt to *compensate* for the changed computation, allowing the model to maintain its intended task performance on the affected platform. This behavior arises from the adaptive nature of learning, where the model state continuously evolves under the computation used for optimization. Consequently, temporary changes in computation semantics can be internalized through the learned weights, leaving persistent effects without necessarily causing an obvious degradation in training or validation performance.

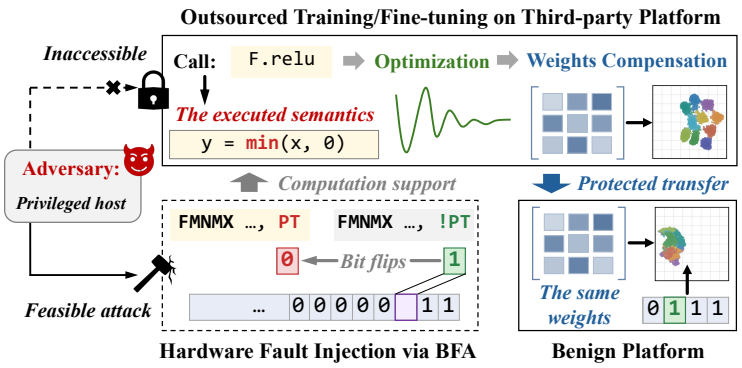


Figure 1: Illustration of Exploiting Optimization via BFAs.

This learning behavior creates a new attack opportunity for an untrusted training host. As shown in Fig. 1, by altering the underlying computation through bit-flip attacks (BFAs), the host can exploit optimization itself to internalize the fault effects into the trained model, without directly accessing model weights or training data. While the induced hardware faults remain active, the learned weights compensate for the altered computation, allowing both training and validation to appear normal and thereby concealing the ongoing attack. Once training completes and the model is transferred to the user's clean platform, the platform executes the original unmodified computation while the learned compensation remains encoded in the weights. Without the altered computation it was learned to offset, this compensation instead drives the model toward the adversarial abnormal behavior. The attack effect therefore becomes observable only after the attack has already completed, making it difficult to detect through conventional validation performed on the compromised training platform.

We refer to this new attack paradigm as *fault internalization*. To understand how it can be realized in practice, we systematically study code-level bit flips across different training settings and models. This requires identifying target bits whose flips induce valid (e.g., preserving valid instruction decoding and alignment while avoiding illegal memory accesses) and stable alternative computation semantics without making training visibly abnormal, and understanding how these altered computations can repeatedly influence optimization and ultimately be internalized into the learned weights to produce adversarial model-level effects. Through this study, we identify recurring characteristics of effective bit flips and distill them into methodological guidance and heuristics for locating target bits in the underlying computation libraries.

Using the identified target bits, we evaluate fault internalization across image classifiers, LLMs, CVAEs, and classifier-guided Diffusion under training-from-scratch, full-weight fine-tuning, and LoRA fine-tuning. Across these settings, optimization can absorb the altered computation into the learned weights: models under attacked execution retain benign-like task performance and stable training behavior, including cases where pretrained representations must re-adapt to the altered computation or where compensation is confined to LoRA adapter weights. Once the same checkpoints are transferred to a benign platform, restoring the original computation exposes the internalized alteration, causing fixed or near-random classifications, degraded LLM decisions, semantically coherent but incorrect generations, and severe failures in image generation. Our experiments further show that fault internalization follows the affected execution path and can accumulate across iterative inference, revealing how the same mechanism manifests differently across model structures and inference procedures. The evaluated alterations require only 3-40 bit flips in the corresponding computation-library implementations. We also discuss mitigation strategies for reducing the risk of

fault internalization in outsourced training environments, as well as its broad implications to training reliability. Overall, this paper makes the following contributions:

- Conceptually, we identify and formulate *fault internalization*, where optimization absorbs altered computation into the learned weights. This enables an untrusted training host to conceal an ongoing BFA during training and expose its effect only when the delivered model returns to the original computation.
- Technically, we characterize the semantic- and implementation-level conditions that enable fault internalization and develop a systematic procedure for mapping suitable computation alterations to target bits in underlying computation libraries.
- Empirically, we evaluate fault internalization across discriminative and generative models under training-from-scratch, full-weight fine-tuning, and LoRA fine-tuning. With only 3-40 bit flips, attacked execution remains close to benign behavior, whereas restoring the original computation exposes diverse failures that follow the affected execution path and can accumulate across iterative inference.

2 Background and Related Work

Outsourced Training and Protections. AI model training is increasingly carried out on outsourced or externally managed computing infrastructures, where model owners rely on third-party resources to train or fine-tune their models Hunt et al. (2018); Zhang et al. (2021). To protect sensitive training data and proprietary model states in such environments, prior systems have explored trusted execution environments (TEEs) Hunt et al. (2018); Kunkel et al. (2019); Zhang et al. (2021) and cryptographic protocols such as secure multi-party computation Mohassel & Zhang (2017); Mohassel & Rindal (2018). These mechanisms restrict direct exposure of training data and model states, yet the training process still relies on the underlying software and hardware stack to execute model computation. In modern learning frameworks, high-level operators are dispatched through framework runtimes and optimized libraries to hardware-accelerated implementations, whose forward and backward computations jointly determine the gradients used for weight updates Paszke et al. (2019). Therefore, in addition to the protected data and weights, the resulting model also depends on the computational semantics provided by the training stack.

Bit-Flip Attacks & Training Reliability. Prior BFAs against AI models primarily manipulate the binary representation of model weights, often using gradient-guided search to identify vulnerable bits Rakin et al. (2019; 2021); Bai et al. (2021); Chen et al. (2021a). Recent work extends weight-level BFA to large language models (LLMs), degrading task performance while preserving natural-looking outputs Xu et al. (2026). Training-assisted BFA further incorporates model training to construct a high-risk model, where flipping a critical weight bit in the deployed model triggers targeted misclassifications Dong et al. (2023). Training in this setting prepares a vulnerable weight configuration, and ***the attack remains triggered by post-training modification*** of the learned weights. These attacks still rely on access to model weights or gradients, which are unavailable in protected outsourced training. Code-level BFAs avoid weight manipulation by altering executable instructions, but existing attacks mainly disrupt computation or instruction execution Chen et al. (2025a); Li et al. (2024), making them unsuitable for training that requires viable forward and backward computation. Fault-robustness training instead treats hardware faults as execution errors to tolerate or correct Zahid et al. (2020); Yu et al. (2023). Our work studies code-level BFAs that locally alter operator semantics without disrupting optimization, allowing the learned weights to internalize the altered computation and expose abnormal behavior only after the original semantics are restored.

3 Problem Formulation

3.1 Problem Setting and Threat Model

Problem Setting. We consider outsourced model training and fine-tuning on third-party computing platforms. Since the platform host may be untrusted yet retains high system privileges, the training workload is often protected by a trusted execution environment (TEE), which isolates models, training data, and runtime states from direct access or modification by the host. Remote attestation further allows the user to verify that the workload is executed on a genuine platform with the

expected software stack and configuration. Because the underlying hardware and software execution environment is provided by the third-party platform, the host is aware of their implementation details; however, it cannot directly modify the attested software or hardware state.

Adversary's Capabilities and Knowledge. The adversary (i.e., the host) by default has full knowledge of the software stack provided by the platform, including the computation libraries and their compiled binary code. The host can therefore analyze the corresponding source code, binaries, and disassembly. Nevertheless, the TEE prevents the host from directly accessing protected model weights, training data, and runtime states, or computing gradients over the protected weights, rendering most existing weight-level BFAs inapplicable. The host can leverage well-developed hardware fault-injection techniques (e.g., RowHammer) to induce targeted bit flips during execution. Combined with its knowledge of the computation implementation, this capability makes its binary code a practical target for BFAs even when the model weights and training data remain protected.

Fault Model. We model each fault as a bit flip in the compiled binary code of the computation libraries during training. A bit flip changes an existing bit from 0 to 1 or from 1 to 0, and the adversary cannot arbitrarily rewrite the binary code or insert and remove bits. Using its knowledge of the computation implementation, the adversary can determine the target bits in advance and induce the corresponding flips during execution.

Attack Objective. A successful training-stage BFA should satisfy three objectives. First, both training and the same-platform validation should remain normal during fault injection. Second, the fault effect should be carried into the learned weights through training, so that the post-training effect emerges only after the model is transferred to the user's clean platform, where it exhibits the adversarial behavior. Third, the attack should operate without accessing or modifying model weights and training data. Achieving these objectives requires a fundamentally different type of bit flip from the destructive faults used by existing library-level BFAs: the bit flips should induce a suitable computation ***alteration*** that remains compatible with normal optimization and allows its effect to be absorbed through training.

3.2 Characteristics of Internalizable Computation Alterations

Given the above setting, a key question is which library-level faults can persist through training and become internalized by the model. We characterize this from two perspectives. At the semantic-Level, the computation alteration should preserve ***structural learnability***, ***weight compensability***, and ***optimization viability***. At the implementation-level, it should also preserve ***executability***, ***locality***, and ***forward-backward compatibility***. More details are given in Appx. E due to limited space.

4 Methodology

4.1 From High-Level Operations to Binary Bits

Candidate Operator Identification. We begin from the training computation graph and screen candidates at the operator level. We focus on activation and pooling operators because they modify intermediate features through explicit, parameter-free computation rules. Their semantics are separated from the model's learned weights: changing the operator alters the computation applied to activations without directly modifying the surrounding weights. This makes them suitable targets for studying whether optimization can absorb systematic changes in computation semantics. Their pointwise or bounded-window dependencies also lead to well-defined forward and backward computations, which facilitates controlled modification of both forward and backward propagation.

Characteristic-Guided Semantic Alteration. Guided by the semantic-level characteristics in Sec. 3.2, we seek alterations that change an operator's numerical mapping while preserving its computational scaffold, including tensor dependencies, interfaces, and forward–backward structure. This narrows the search from arbitrary operator modifications to ***local semantic relations***. Across the selected activation and pooling operators, we identify two recurring forms: ***(1) selection relations***, such as piece-wise branch or max/min choices, and ***(2) sign relations*** that determine the sign of operands or intermediate terms. We use these relations as semantic alteration candidates for subsequent implementation mapping.

Cross-Layer Implementation Mapping. The semantic alterations identified above are defined over high-level operators, whereas their native implementations depend on the execution path selected by the target workload. Moreover, tensor shape, dtype, and layout may route the same operator through different backend paths, and its forward and backward computations may reside in distinct routines. Therefore, for each candidate operator, we trace the actual forward and backward calls from the Python functions, through `libtorch_cuda.so`, to the launched CUDA kernels. We record the corresponding tensor metadata and track forward-saved values or indices consumed during backpropagation to resolve dependencies between the two paths. We then extract the architecture-specific CUDA binary and disassemble it into SASS (i.e., NVIDIA's native GPU instructions). Finally, we follow the comparison, selection, and arithmetic dataflow to localize the native instructions that implement the targeted semantic relation.

Implementation-Level Realization. Given the localized native instructions, we seek a bit-level realization of the targeted semantic alteration under the fault model defined in Sec. 3.1. For each workload-reachable path, we obtain the instruction offsets and raw encodings. We construct controlled instruction variants that differ only in the targeted condition, selection relation, or operand sign, and align their corresponding native instructions. We then compute the bitwise XOR of their encodings to identify the bit positions that realize the semantic difference.

Guided by the implementation-level characteristics in Sec. 3.2, we alter only the local computation that implements the targeted semantic relation, with the surrounding execution structure and data dependencies preserved. This confines the induced semantic change to the mapped computation. We construct the forward and backward realizations jointly: shared implementations use the same local alteration, whereas separate implementations are altered consistently to preserve the altered forward–backward relation. We also apply the alteration across all workload-reachable paths that implement the same semantic relation. We then re-disassemble the altered kernel and compare its forward outputs and input gradients against a reference implementation of the altered semantics to verify the selected bits. The original operator interface and dispatch path remain unchanged, allowing calls mapped to the same backend routine to repeatedly execute the altered instructions across layers and optimization steps, without requiring separate alterations for individual invocations.

4.2 Attack Instantiations

This section instantiates the above procedure. Due to the limited space, we only show the ReLU cases; more examples are given in Appx. B. We further extend the same procedure to LoRA, a widely used process for fine-tuning LLMs through trainable low-rank adaptation branches.

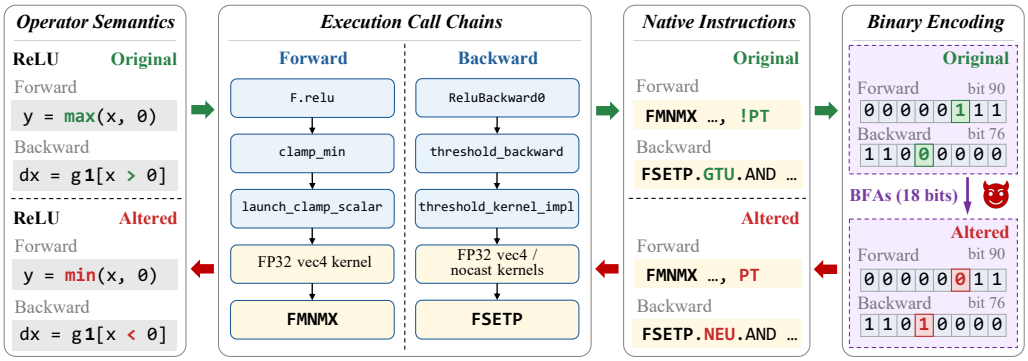


Figure 2: Attack Instantiation on ReLU. The attacker only flips bits in the binary encoding via BFAs, without modifying call chains or Python code. The attack takes effect from right to left.

ReLU. Fig. 2 shows the ReLU alteration due to bit flips in its binary encoding. We replace the positive activation region with the negative region and change the backward gate accordingly:

$$\widetilde{y} = \min(x, 0), \qquad \frac{\partial \mathcal{L}}{\partial x} = g, \mathbf{1}[x < 0]. \tag{1}$$

The forward selection and backward gate are implemented separately through `FMNMX` and `FSETP`, respectively. Since the backward computation gates the upstream gradient using the saved forward

output, retaining the original positive-side comparison would suppress gradients on the altered active region. We therefore modify the backward comparison to remain consistent with the new forward semantics. The altered operator remains piecewise linear and preserves a unit derivative on its active region, allowing optimization to continue under the modified computation. The forward and backward realizations require 8 and 10 bit flips, respectively.

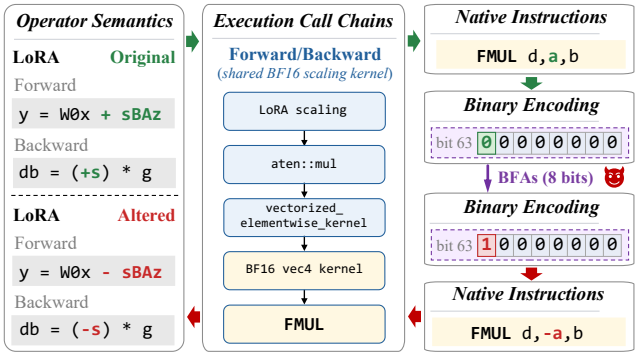


Figure 3: Attack Instantiation on LoRA Computation.

LoRA Computation. In addition to generic activation and pooling operators, we apply the same procedure to the scaling computation introduced by LoRA fine-tuning, as illustrated in Fig. 3. For an input x, the standard computation is $W_0x + sBAz$. We change only the adapter contribution to obtain $W_0x - sBAz$, leaving the frozen base branch and the stored scale unchanged. Under BF16 autocast, the forward and backward computations use the same tensor scalar multiplication kernel. Flipping the source negation bit in four body and four tail `FMUL` instructions applies the negative scale in both directions without changing the matrix multiplications. Since only the adapter parameters are trainable, compensation must be expressed through the adaptation branch. With zero-initialized B, the negative scale can be absorbed by learning an adapter contribution with the opposite sign, preserving the model output under the altered computation. Appx. G proves that, once this compensation relation is established, the attacked execution exactly reproduces the corresponding benign outputs, logits, and losses throughout training in exact arithmetic. The shared forward and backward realization requires 8 bit flips in total.

5 Evaluation

5.1 Setup

Configurations. We evaluate fault internalization under three execution conditions: ① ***Benign*** trains or fine-tunes the model and evaluates the resulting checkpoint under the original (i.e., benign and unaltered) computation; ② ***Attacked*** keeps the injected bit flips active throughout the outsourced training or fine-tuning process and evaluates the resulting checkpoint under the same altered computation; ③ ***Restored*** transfers the checkpoint obtained under the ***Attacked*** condition to a clean platform and evaluates it under the original computation without any further weight update. The three conditions use the ***same*** data, training schedule, and evaluation protocol, and differ only in the computation that is induced by flipping binary bits via BFAs. All experiments are conducted on one NVIDIA RTX A6000 GPU using PyTorch 2.5.1 and CUDA 12.1.

Attack Realization. We realize each alteration solely by flipping the identified bits (see Sec. 4.2 and Appx. B) in the compiled computation-library binary. All other component, including the Python code, framework operators, dispatch interfaces, training procedure, et.c, remain unchanged, such that all executions continue through the original software stack.

Evaluation Organization. The following experiments aim to answer two research questions:

▷ ***RQ1*** (Mechanism; Sec. 5.2): *How does optimization internalize altered computation?*

▷ ***RQ2*** (Generality; Appx. C): *How does fault internalization generalize across computation functions, tasks, modalities, and training settings?*

5.2 Fault Internalization and Its Mechanisms (RQ1)

To answer RQ1, we focus on image classifiers trained from scratch, which provides a controlled setting for comparing different computation alterations, model architectures, and datasets under the

same training objective. We examine their task performance, training dynamics, decision functions, and intermediate representations to study how optimization internalizes the altered computation. Unless otherwise specified (see details in Appx. F), all models are trained in FP32 with a batch size of 128 using SGD. The learning rate starts at 0.05 and follows a cosine decay schedule. Appx. D further assesses fault internalization with different optimizers, learning rates, and batch sizes, where it remains consistently effective.

Table 1: Test accuracy (%) under benign, attacked, and restored execution for image classifiers trained from scratch. Values are mean ± sample standard deviation over three runs with different seeds. For LeNet/MNIST, the MaxPool results use the LeNet variant with MaxPool layers. Restored outcomes are categorized as ▢ systematic misclassification, ▢ fixed-class predictions, ▢ random predictions, and ▢ mixed fixed-class/random outcomes across runs.

Model / Dataset	ReLU			MaxPool			AvgPool		
	Benign	Attacked	Restored	Benign	Attacked	Restored	Benign	Attacked	Restored
LeNet MNIST	99.16 ± 0.02	98.57 ± 0.16	12.72 ± 1.98	99.16 ± 0.02	98.61 ± 0.05	47.14 ± 12.66	98.95 ± 0.03	98.98 ± 0.07	6.17 ± 3.02
ResNet-18 CIFAR-10	88.45 ± 0.31	88.36 ± 0.40	10.14 ± 0.24	88.45 ± 0.31	88.26 ± 0.12	10.95 ± 2.38	88.70 ± 0.04	88.62 ± 0.11	0.04 ± 0.02
VGG-11 CIFAR-10	91.60 ± 0.13	87.87 ± 0.19	9.76 ± 0.38	91.60 ± 0.13	87.77 ± 0.07	12.23 ± 2.05	91.65 ± 0.16	91.56 ± 0.09	1.15 ± 0.22
DenseNet-121 CIFAR-10	90.35 ± 0.50	89.94 ± 0.14	10.00 ± 0.01	90.35 ± 0.50	90.06 ± 0.20	10.05 ± 0.13	90.35 ± 0.50	90.58 ± 0.24	5.30 ± 3.47
Inception-v3 CIFAR-10	93.34 ± 0.12	92.57 ± 0.30	10.00 ± 0.00	93.34 ± 0.12	92.68 ± 0.22	10.47 ± 0.85	93.34 ± 0.12	93.55 ± 0.08	10.00 ± 0.00
ResNet-50 CIFAR-100	63.35 ± 0.50	63.64 ± 0.54	1.00 ± 0.00	63.35 ± 0.50	63.36 ± 0.67	2.06 ± 0.16	63.24 ± 1.29	62.72 ± 1.15	0.00 ± 0.00
VGG-16 CIFAR-100	72.31 ± 0.19	67.24 ± 0.20	0.98 ± 0.03	66.76 ± 0.17	58.37 ± 0.27	1.80 ± 0.47	72.13 ± 0.10	71.89 ± 0.28	0.013 ± 0.006

Attack Effectiveness. We first evaluate whether fault internalization can preserve normal task performance during attacked training (i.e., bit-flip faults injected via BFAs) and induce substantial model degradation after the original computation is restored. Table 1 reports the Benign, Attacked, and Restored accuracies for ReLU, MaxPool, and AvgPool across different models and datasets. In 17 of the 21 configurations, Attacked accuracy differs from Benign accuracy by less than 1%, showing that optimization can preserve the original task behavior under BFA-altered computation semantics. Once the original computation is restored (e.g., on the user's own platform), all evaluated configurations exhibit substantial degradation. The resulting predictions exhibit different failure patterns: some are ***pinned to a fixed class*** (▢), some become ***random predictions*** (▢), and several AvgPool cases ***fall substantially below random accuracy*** (▢). For ResNet, the attacked AvgPool directly precedes the final linear classifier, allowing its sign change to be absorbed by the classifier weights during training. Restoring the original AvgPool can then approximately reverse the learned class-score ordering when the classifier bias is small, producing systematic misclassification instead of random guessing. Since Attacked and Restored use the same learned weights, these changes arise from executing the learned weights under the restored computation.

Compensation Behavior. The degree of compensation in Table 1 varies across the three alterations and model architectures. AvgPool gives the most consistent Attacked performance, with all evaluated models remaining close to their Benign accuracy. This is because the sign alteration preserves the pooled information and gradient magnitude, allowing nearby weights to absorb the changed computation directly. Such a direct absorption also strengthens the effect of fault internalization, as more complete compensation under Attacked execution can lead to larger degradation after restoration. ReLU and MaxPool also retain Benign-level performance on most architectures; nevertheless, their VGG cases show modest degradation. Unlike ResNet, DenseNet, and Inception, VGG propagates features through a sequential stack without residual or parallel paths, such that changes to activation or selection semantics affect the subsequent computation more directly and require stronger compensation during training. Most Restored models lose nearly all classification capability, whereas MaxPool on LeNet retains 47.14% accuracy with substantially larger variation

across runs. MaxPool changes the selected response in each pooling window, without changing the window or downsampling structure, and the shallow LeNet on MNIST can retain part of the learned class information after maximum selection is restored.

Training Dynamics. Fig. 4 compares the training loss of LeNet and ResNet-18 under Benign and Attacked execution. Across ReLU, MaxPool, and AvgPool, the Attacked runs follow similar convergence trajectories to their Benign counterparts, without introducing persistent oscillation, divergence, or stalled optimization. The two curves are particularly close under AvgPool, whereas ReLU and MaxPool introduce modest but still limited deviations during training. These results show that the altered computations preserve the optimization signals required for learning, allowing the attack to remain inconspicuous under conventional monitoring of training loss. Due to the space limit, we provide the loss curves of other cases in Appx. H.

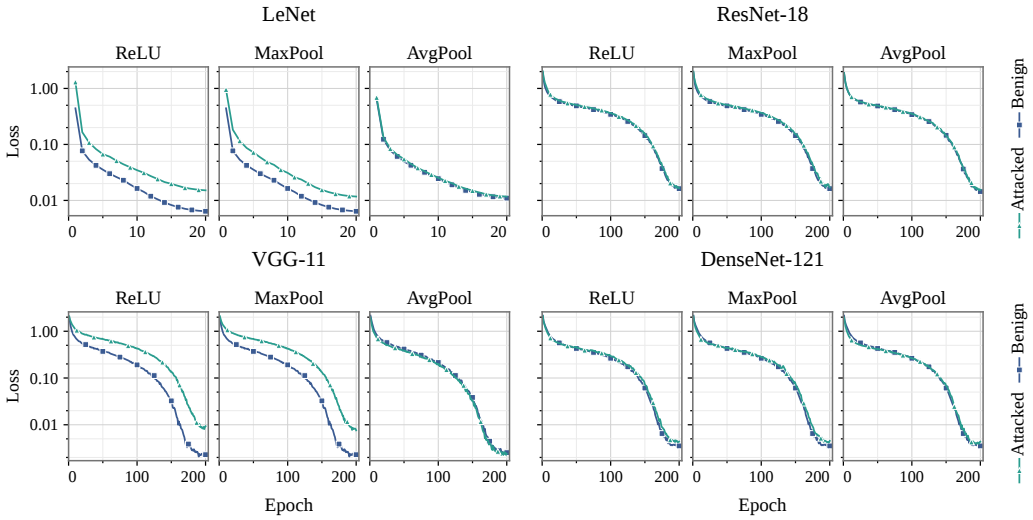


Figure 4: Training loss curves for LeNet, ResNet-18, VGG-11, and DenseNet-121; more curves are provied in Appx. H. Loss is shown on a logarithmic scale to improve visibility.

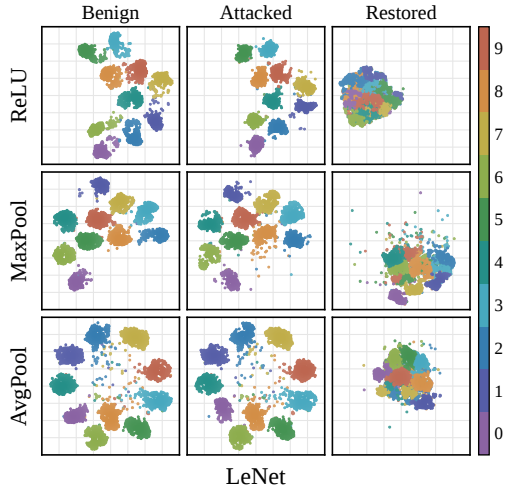


Figure 5: t-SNE visualization of LeNet logits.

Decision-Level Compensation. We also examine how compensation appears at the model output. Fig. 5 compares the logit structures under Benign, Attacked, and Restored execution. Benign and Attacked produce similar class organization in the shared embedding, suggesting that optimization adjusts the weights to reconstruct the discriminative output structure under the attack-altered computation. This compensation extends beyond preserving the final prediction, as the distribution and relative separation among classes is also retained by comparing "Benign" and "Attacked" columns. After restoration, the same attack-trained weights operate with the original ReLU, leaving the learned compensation unmatched by the computation it was formed to offset. The resulting logits become concentrated in an overlapping region, disrupting class separation across the output space. We give more cases in Appx. I.

Feature-Distribution Alignment. We further examine whether the compensation formed during Attacked training keeps downstream representations aligned with the Benign feature distribution. Following the common practice in anomaly detection, we train a one-class autoencoder for each monitored layer using only Benign features. Then, we use its reconstruction error to distinguish Attacked from Benign representations; more details are provided in Appx. F. As shown in Table 2,

across all monitored layers, Attacked features produce reconstruction errors close to those of Benign features, and their alarm rates ($0.005\% \sim 1.225\%$) remain around or below the Benign false-positive rates ($0.97\% \sim 1.43\%$). Moreover, while there is a weak difference (AUROC of 0.575) observed immediately after the altered MaxPool, it does not persist through the downstream network, and the AUROC falls to $0.409 \sim 0.500$ in the subsequent layers. The downstream computation progressively absorbs the representation change introduced by the altered pooling semantics, bringing the Attacked features back toward the Benign feature distribution. These results indicate that fault internalization preserves feature-level normality across the downstream representations.

Table 2: Feature-level anomaly analysis for ResNet-18 on CIFAR-10 under the MaxPool attack. Reconstruction errors are normalized. AUROC measures the separability between Benign and Attacked representations.

	Reconstruction Error (MSE)		Alarm Rate (%)		AUROC
Layer	Benign	Attacked	Benign	Attacked	Benign vs. Attacked
MaxPool	0.317	0.315	1.11	0.005 ± 0.007	0.575
Layer +1	0.371	0.334	0.97	0.045 ± 0.021	0.471
Layer +2	0.506	0.457	1.43	0.635 ± 0.247	0.409
Layer +3	0.479	0.479	1.01	1.225 ± 0.431	0.500
Layer +4	0.582	0.572	1.09	0.755 ± 0.233	0.476
Penultimate	0.582	0.572	1.09	0.755 ± 0.233	0.476

6 Mitigation and Discussion

Mitigation for Fault Internalization. Existing BFA defenses mainly protect corrupted weights, detect abnormal behavior, or improve robustness under faulty execution He et al. (2020); Özdenizci & Legenstein (2022); Kummer et al. (2025); Chen et al. (2025c). Fault internalization instead alters training computation and can remain concealed under Attacked execution. Even executable-level protection such as BitShield Chen et al. (2025b) assumes fixed model weights, making its semantic references difficult to maintain during optimization. We suggest training-time execution integrity, independent clean validation, and execution-aware attestation as complementary mitigations. Appx. A.1 provides detailed discussion.

Broader Implications. Fault internalization shows that successful optimization and benign-like validation do not necessarily imply reliable training when executed computation deviates from its intended semantics. This may also matter for silent computation corruption in large-scale AI training, where persistent structured faults could be absorbed by optimization without visible instability Ma et al. (2025); Altenbernd et al. (2026); Tyagi et al. (2026). More broadly, learned weights can become coupled to their training-time computation semantics, affecting model portability and reproducibility. Appx. A.2 discusses these implications further.

Limitations and Future Work. Our study covers selection and sign relations across activation, pooling, attention normalization, and LoRA scaling, yet other computation semantics may also support fault internalization. Target-bit identification currently requires manual analysis of operator semantics and native instructions, as is common in code-level BFAs Chen et al. (2025a). Once identified, however, the same backend targets can be reused across models, layers, and training runs. Future work can expand the semantic scope and automate this analysis. See Appx. A.3 for details.

7 Conclusion

We uncover *fault internalization*, where optimization absorbs BFA-induced computation alterations into the learned weights, allowing the attack to remain concealed during training and emerge after the original computation is restored. We characterize the semantic and implementation conditions that enable this behavior and develop a systematic procedure for identifying target bits in computation libraries. Across discriminative and generative models under training-from-scratch, full-weight fine-tuning, and LoRA fine-tuning settings, attacked execution can remain close to benign behavior, whereas restoration exposes diverse model-level failures. These results reveal a new interaction between hardware faults and optimization, motivating the protection of computation semantics.

AI USE STATEMENT

Generative AI tools were used to assist with language editing. They were not used to generate experimental data. All AI-assisted content was reviewed and verified by the authors, who take full responsibility for the final text, technical claims, analyses, and conclusions.

ETHICS STATEMENT

This work studies the security and reliability implications of bit-flip faults during outsourced AI model training. As with other research on fault injection and adversarial attacks, the techniques investigated in this paper have potential dual-use implications. Our purpose is to characterize a previously overlooked failure mode and inform the design of more reliable protections for training platforms. The experiments are conducted in controlled research environments and do not target third-party systems or live services. The study does not involve human subjects or the collection of personal or sensitive user data. We report the assumptions, capabilities, and limitations of the attack explicitly and discuss mitigation strategies, including training-time execution integrity, independent clean validation, and execution-aware attestation. These practices are intended to support responsible disclosure, reproducibility, and the avoidance of unintended harm, consistent with the principles of the ICLR Code of Ethics.

REPRODUCIBILITY STATEMENT

We provide an anonymized code repository at `https://anonymous.4open.science/r/fault-internalization-138A`, which contains the attack implementations, training and evaluation scripts, and experiment configurations used in this work.

REFERENCES

Anton Altenbernd, Philipp Wiesner, and Odej Kao. Exploring silent data corruption as a reliability challenge in llm training. *arXiv preprint arXiv:2604.00726*, 2026.

Jiawang Bai, Baoyuan Wu, Yong Zhang, Yiming Li, Zhifeng Li, and Shu-Tao Xia. Targeted attack against deep neural networks via flipping limited weight bits. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=iKQAk8a2kM0`.

Pierre Carru. Attack arm trustzone using rowhammer. In *GreHack*, 2017. URL `https://grehack.fr/2017/program/`.

Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Proflip: Targeted trojan attack with progressive bit flips. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 7698–7707. IEEE, 2021a.

Yanzuo Chen, Zhibo Liu, Yuanyuan Yuan, Sihang Hu, Tianxiang Li, and Shuai Wang. Compiled models, built-in exploits: Uncovering pervasive bit-flip attack surfaces in dnn executables. In *32nd Network and Distributed System Security Symposium (NDSS)*, 2025a.

Yanzuo Chen, Yuanyuan Yuan, Zhibo Liu, Sihang Hu, Tianxiang Li, and Shuai Wang. Bitshield: Defending against bit-flip attacks on dnn executables. *computing*, 2:47, 2025b.

Yuhang Chen, Zhen Tan, Ajay Kumar Jaiswal, Huaizhi Qu, Xinyu Zhao, Qi Lin, Yu Cheng, Andrew Kwong, Zhichao Cao, and Tianlong Chen. Bit-flip error resilience in llms: A comprehensive analysis and defense framework. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 10425–10435, 2025c.

Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. VoltPillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 699–716, 2021b.

Kamran Chitsaz, Goncalo Mordido, Jean-Pierre David, and François Leduc-Primeau. Training dnns resilient to adversarial and random bit-flips by learning quantization ranges. *Transactions on Machine Learning Research*, 2023.

Jianshuo Dong, Han Qiu, Yiming Li, Tianwei Zhang, Yuanjie Li, Zeqi Lai, Chao Zhang, and Shu-Tao Xia. One-bit flip is all you need: When bit-flip attack meets model training. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 4665–4675. IEEE, 2023.

Zhezhi He, Adnan Siraj Rakin, Jingtao Li, Chaitali Chakrabarti, and Deliang Fan. Defending and harnessing the bit-flip based adversarial weight attack. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 14083–14091. IEEE, 2020.

Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint arXiv:1803.05961*, 2018.

Troya Çağıl Köylü, Cezar Rodolfo Wedig Reinbrecht, Said Hamdioui, and Mottaqiallah Taouil. Deterministic and statistical strategies to protect anns against fault injection attacks. In *2021 18th International Conference on Privacy, Security and Trust (PST)*, pp. 1–10. IEEE, 2021.

Lorenz Kummer, Samir Moustafa, Wilfried Gansterer, and Nils Morten Kriege. Crossfire: an elastic defense framework for graph neural networks under bit flip attacks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 17990–17998, 2025.

Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Tensorscone: A secure tensorflow framework using intel sgx. *arXiv preprint arXiv:1902.04413*, 2019.

Shaofeng Li, Xinyu Wang, Minhui Xue, Haojin Zhu, Zhi Zhang, Yansong Gao, Wen Wu, and Xuemin Sherman Shen. Yes,One-Bit-Flip matters! universal DNN model inference depletion with runtime code fault injection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 1315–1330, 2024.

Jeffrey Jian Ma, Hengzhi Pei, Leonard Lausen, and George Karypis. Understanding silent data corruption in llm training. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 20372–20394, 2025.

Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 35–52, 2018.

Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pp. 19–38. IEEE, 2017.

Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1466–1482. IEEE, 2020.

Ozan Özdenizci and Robert Legenstein. Improving robustness against stealthy weight bit-flip attacks by output code matching. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 13378–13387. IEEE, 2022.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, Ruidong Tian, Chunlu Wang, and Gang Qu. Voltjockey: A new dynamic voltage scaling-based fault injection attack on intel sgx. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(6):1130–1143, 2020.

Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 1211–1220. IEEE, 2019.

Adnan Siraj Rakin, Zhezhi He, Jingtao Li, Fan Yao, Chaitali Chakrabarti, and Deliang Fan. T-bfa: Targeted bit-flip adversarial weight attack. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):7928–7939, 2021.

Abhishek Tyagi, Saurabh Hukerikar, Nirmal Saxena, Yanxiang Huang, Philip Shirvani, Chung-Hsuan Tung, and Yuhao Zhu. Llm-prism: Characterizing silent data corruption from permanent gpu faults in llm training. *arXiv preprint arXiv:2604.10390*, 2026.

Haotian Xu, Qingsong Peng, Jie Shi, Huadi Zheng, Yu Li, and Cheng Zhuo. Silentstriker: Toward stealthy bit-flip attacks on large language models. *Advances in Neural Information Processing Systems*, 38:51981–52002, 2026.

Anlan Yu, Ning Lyu, Jieming Yin, Zhiyuan Yan, and Wujie Wen. Cola: Orchestrating error coding and learning for robust neural network inference against hardware defects. In *International Conference on Machine Learning*, pp. 40277–40289. PMLR, 2023.

Ussama Zahid, Giulio Gambardella, Nicholas J Fraser, Michaela Blott, and Kees Vissers. Fat: Training neural networks for reliable inference under hardware faults. In *2020 IEEE International Test Conference (ITC)*, pp. 1–10. IEEE, 2020.

Chengliang Zhang, Junzhe Xia, Baichen Yang, Huancheng Puyang, Wei Wang, Ruichuan Chen, Istemi Ekin Akkus, Paarijaat Aditya, and Feng Yan. Citadel: Protecting data privacy and model confidentiality for collaborative learning. In *Proceedings of the ACM symposium on cloud computing*, pp. 546–561, 2021.

A Detailed Discussion

A.1 Mitigation for Fault Internalization

A.1.1 Why Existing BFA Defenses Fall Short

Existing defenses against BFAs on AI models mainly protect corrupted model states, detect abnormal model behavior, or improve robustness to fault-induced perturbations. Fault internalization alters the training computation and allows optimization to absorb the alteration, leaving these defense principles insufficient by design.

Weight-Centric Protection. Weight-centric defenses protect or recover corrupted bits in learned weights through integrity checking, selective bit protection, or bit-level correction Kummer et al. (2025); Chen et al. (2025c). Fault internalization does not inject faults into the learned weights. The BFA instead alters the computation-library implementation, and the weights are subsequently updated through the normal optimization process. The delivered checkpoint thus contains no attacker-flipped weight bits for these defenses to detect or correct.

Behavior-Based Detection. Other defenses detect fault injection through changes in model outputs or intermediate activations Köylü et al. (2021), and output-oriented defenses reduce attack stealthiness by making weight faults more visible in model behavior Özdenizci & Legenstein (2022). Fault internalization violates this detection assumption because optimization continuously re-adapts the weights to the altered computation. Under Attacked execution, the model can retain task performance and intermediate representations close to Benign execution (Sec. 5.2). The model-level effect becomes exposed after the original computation is restored, when the injected faults are no longer active. Monitoring model behavior only under the altered execution can therefore miss the attack.

Fault-Robust Training. Training-based defenses improve model tolerance to bit-flip perturbations through robust weight representations, learned quantization ranges, or fault-aware fine-tuning He et al. (2020); Chitsaz et al. (2023); Chen et al. (2025c). Their objective is to preserve model performance under faulty execution. Fault internalization instead exploits optimization's ability to adapt to a persistent alteration in computation semantics. Maintaining performance under the altered computation can therefore coexist with successful internalization, without ensuring that the learned weights remain compatible with the original computation.

Executable-Level Protection. BitShield extends BFA defense to executable code Chen et al. (2025b). However, it does not directly cover training because its protection is built around a fixed

model weights. BitShield checks runtime semantics against a precomputed reference and couples code checksums with statically transformed weights. Directly applying this design to training is difficult because optimization continuously updates the weights, invalidating the semantic reference and requiring the checksum-coupled transformation to change accordingly. Such an extension must also distinguish legitimate semantic changes caused by optimization from those introduced by BFAs.

A.1.2 Suggested Mitigation

Our findings suggest that protecting outsourced training requires integrity guarantees over the computation that produces the learned weights, in addition to the model state and training data. We discuss three complementary directions covering training-time execution integrity, independent clean validation, and execution-aware attestation of the training process.

Training-Time Execution Integrity. A direct mitigation is to protect the computation libraries and device kernels exercised during training, rather than only the weights and data they process. Integrity checking should cover both forward and backward paths because fault internalization relies on consistent semantic alterations across optimization. Beyond binary integrity, critical operator outputs and input gradients can be selectively compared against trusted reference implementations to verify the executed semantics. Such checks should rely on training-invariant computation relations or code integrity, instead of fixed model behavior that naturally evolves as the weights are updated.

Independent Clean Validation. Validation performed on the training platform may share the same altered computation that conceals fault internalization. Before accepting a delivered checkpoint, the user should therefore re-evaluate it on an independent clean platform under the intended computation semantics, without further weight updates. This directly tests whether the learned weights remain compatible with the original computation and can expose compensation that was hidden during outsourced training. Such validation complements training-time integrity checking, although its coverage remains limited to the behaviors exercised by the validation workload.

Execution-Aware Attestation. Outsourced training should also provide verifiable evidence of the computation semantics under which the delivered checkpoint was produced. Fault internalization shows that high-level training code alone does not establish the semantics actually executed by the underlying libraries and kernels. Attestation can therefore bind the checkpoint to verified computation-library binaries, workload-reachable kernel paths, and integrity measurements collected during training. This provides evidence that the learned weights were optimized under the intended computation rather than an altered implementation.

A.2 Broader Implications

Training Reliability. Fault internalization reveals a broader reliability risk in model training: successful optimization and benign-like validation performance do not guarantee that the learned weights remain reliable under the intended computation. Optimization adapts to the computation actually executed during training, allowing persistent semantic deviations to be absorbed into the weights without causing visible training failure. The resulting checkpoint can therefore remain well-performing in the training environment yet exhibit abnormal behavior once the intended computation is restored. This observation suggests that training reliability should account for the integrity of the executed computation in addition to conventional convergence and validation metrics.

Silent Computation/Bit Corruption. Silent data corruption (SDC) has become an increasing reliability concern in large-scale AI training. Recent studies of LLM training show that SDC can perturb intermediate computation and gradients, alter the optimization trajectory, and in some cases cause loss spikes or divergence Ma et al. (2025); Altenbernd et al. (2026); Tyagi et al. (2026). These works primarily characterize SDC through the errors and instability it introduces during optimization. Fault internalization reveals a complementary possibility: when a computation deviation is persistent, structured, and learnable, optimization may absorb its effect without sustained training failure. Although we currently only launched adversarial bit flips, our results raise the question of whether some silent computation faults can remain concealed because training adapts to them.

Weights and Execution Semantics. Fault internalization also shows that learned weights are not always independent of the computation semantics under which they are optimized. Our results on CVAEs and diffusion further show that this dependence follows the affected execution path and

can accumulate when the same computation is repeatedly invoked. Consequently, preserving the same checkpoint does not by itself guarantee identical behavior after the execution environment changes. Reproducibility and model portability should therefore consider the computation semantics that produced the learned weights, instead of treating the checkpoint as a standalone artifact.

A.3 Limitations and Future Work

Scope of Semantic Alterations. Our current study already covers a broad range of computation patterns, including selection and sign relations across activation, pooling, attention normalization, and LoRA scaling. These instantiations involve different numerical mappings and forward–backward dependencies, showing that fault internalization is not tied to a particular operator family or semantic form. Nevertheless, they do not exhaust the space of computation semantics that optimization may internalize. Future work can extend the analysis to additional arithmetic relations, normalization operators, and fused computations, and further characterize which semantic properties make an altered computation amenable to internalization.

Automation of Bit Identification. Unlike weight-level BFAs that can directly search vulnerable bits using model weights and gradients, our current bit identification still involves manual analysis of operator semantics and native instructions. This implementation analysis is also common in existing code-level BFAs, where attack targets must be identified from compiled computation and mapped to the instructions that realize the intended behavior Chen et al. (2025a). That said, the resulting target bits are tied to shared backend implementations instead of individual weight instances, allowing the same alteration to be reused across models, layers, and training runs that exercise the corresponding execution path. This reuse reduces the need to repeat bit search for each model instance after the implementation-level target has been identified. Future work can further automate cross-layer tracing, semantic relation localization, instruction matching, and encoding comparison to reduce the analysis effort while preserving this reuse.

B More Cases of Attack Instantiations

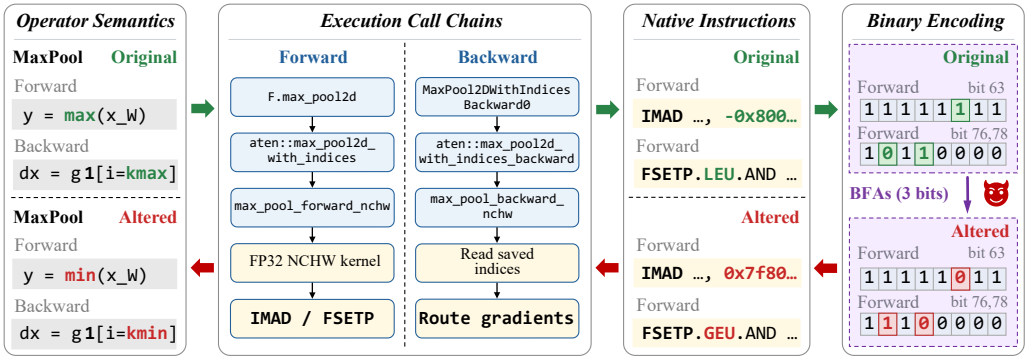


Figure 6: Attack Instantiation on MaxPool. The attacker only flips bits in the binary encoding via BFAs, without modifying call chains or Python code. The attack takes effect from right to left.

MaxPool. In MaxPool, the forward selection determines both the pooled value and the index later used for gradient routing, as illustrated in Fig. 6. For each pooling window W_j, we replace maximum selection with minimum selection and store the corresponding selected index:

$$\widetilde{y}_j = \min_{i \in W_j} x_i, \qquad \widetilde{a}_j = \arg,\min_{i \in W_j} x_i. \tag{2}$$

In the FP32 NCHW kernel, one bit changes the extremum initialization from $-\infty$ to $+\infty$, while two additional bits reverse the comparison from `LEU` to `GEU`, so that the value and index follow the same selection rule. The initialization must change together with the comparison, since a minimum search initialized at $-\infty$ would retain the initial value for finite inputs. The backward kernel then uses the saved index to accumulate upstream gradients at the selected locations, including contributions

from overlapping windows, without requiring a separate change to its gradient-routing logic. The forward realization requires 3 bit flips, and the backward path requires no additional bit flip. After the original MaxPool is restored on a clean platform, the same learned weights receive maximum-selected features instead of the minimum-selected features used during training.

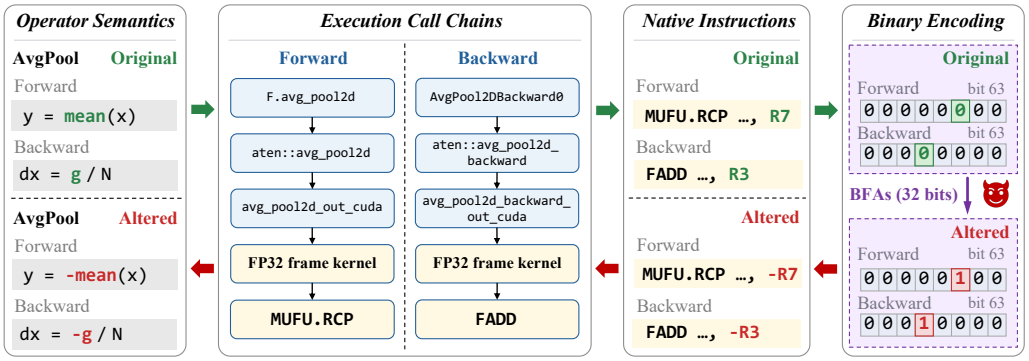


Figure 7: Attack Instantiation on AvgPool. The attacker only flips bits in the binary encoding via BFAs, without modifying call chains or Python code. The attack takes effect from right to left.

AvgPool. As illustrated in Fig. 7, AvgPool provides a sign-relation instantiation in which the pooling structure remains unchanged. We negate the pooled mean and apply the same sign change to its backward computation:

$$\widetilde{y}_j = -\frac{1}{N_j}\sum_{i\in W_j} x_i, \qquad \frac{\partial \mathcal{L}}{\partial x_i} = -\sum_{j:i\in W_j} \frac{g_j}{N_j}. \tag{3}$$

The original window, stride, padding, and divisor remain unchanged. In the native forward implementation, division is realized through a reciprocal approximation followed by residual corrections. Four bit flips change the relevant divisor operands in `MUFU.RCP` and `FCHK` and adjust the associated `FFMA` corrections, producing a consistent negative divisor throughout the division sequence. The backward computation applies the same sign change to the gradient contributions through 28 source-operand modifications. Since the alteration preserves linearity and gradient magnitude, adjacent weights can continue to optimize under the modified computation. The forward and backward realizations require 4 and 28 bit flips, respectively. Restoring the original AvgPool on a clean platform changes the sign of the pooled features and gradients relative to those used during training.

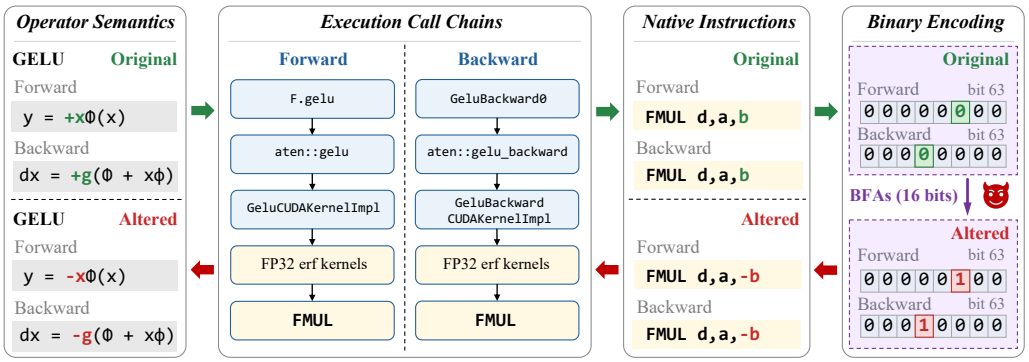


Figure 8: Attack Instantiation on GELU. The attacker only flips bits in the binary encoding via BFAs, without modifying call chains or Python code. The attack takes effect from right to left.

GELU. As shown in Fig. 8, we reverse the output sign of GELU while preserving its input and internal error function computation. Let Φ and ϕ denote the standard normal cumulative distribution

function and density. We change the forward mapping and its corresponding backward rule to

$$\widetilde{y} = -x\Phi(x), \qquad \frac{\partial \mathcal{L}}{\partial x} = -g\big(\Phi(x) + x\phi(x)\big). \tag{4}$$

In the forward path, the final activation value is produced by an FP32 `FMUL`. We flip its source negation bit at four body and four tail sites, requiring 8 bit flips. The backward path applies the same modification to four body and four tail `FMUL` sites, requiring another 8 bit flips. These changes implement $-\mathrm{GELU}(x)$ without modifying the preceding error function computation. The alteration preserves the activation and gradient magnitudes for the same input, allowing adjacent linear weights to absorb the reversed feature polarity during fine-tuning. The complete realization requires 16 bit flips across the forward and backward paths. After the original GELU is restored on a clean platform, the learned weights receive features with the opposite polarity from those used during fine-tuning.

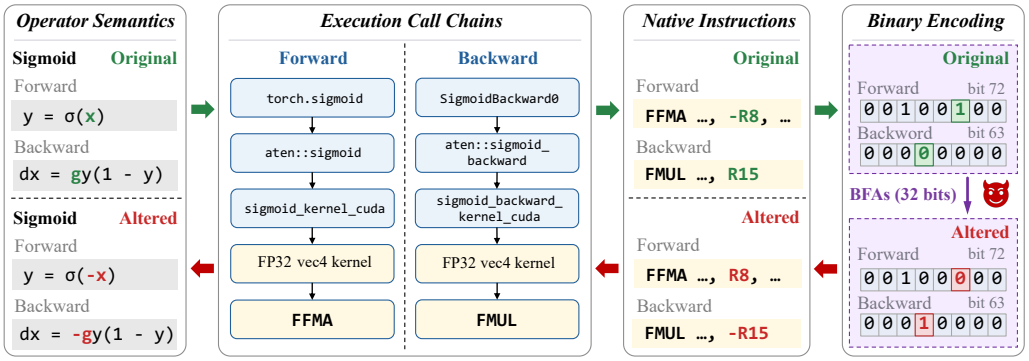


Figure 9: Attack Instantiation on Sigmoid. The attacker only flips bits in the binary encoding via BFAs, without modifying call chains or Python code. The attack takes effect from right to left.

Sigmoid. As illustrated in Fig. 9, we reflect the input of Sigmoid to complement its output while preserving the bounded gating range. With $\sigma(x) = (1 + \exp(-x))^{-1}$, we change the forward mapping and its corresponding backward rule to

$$\widetilde{y} = \sigma(-x) = 1 - \sigma(x), \qquad \frac{\partial \mathcal{L}}{\partial x} = -g\widetilde{y}(1 - \widetilde{y}). \tag{5}$$

In the FP32 forward kernel, the exponential term is computed through range reduction and approximation. We clear the source negation bit in the relevant `FFMA` instructions to replace $-x$ with x throughout this computation. The traced vec4 path contains twelve body and twelve tail sites, requiring 24 bit flips in the forward path. The backward computation uses the saved forward output to form $\widetilde{y}(1 - \widetilde{y})$. We flip the source negation bit in eight final `FMUL` instructions to apply the negative chain-rule factor, requiring 8 additional bit flips in the backward path. The altered computation preserves the gate range in $(0, 1)$ and the gradient magnitude for the same input, allowing optimization to adapt to the complemented gating rule. The complete realization requires 32 bit flips across the forward and backward paths. After the original Sigmoid is restored on a clean platform, the learned weights apply $\sigma(x)$ instead of the complementary gate $\sigma(-x)$ used during fine-tuning.

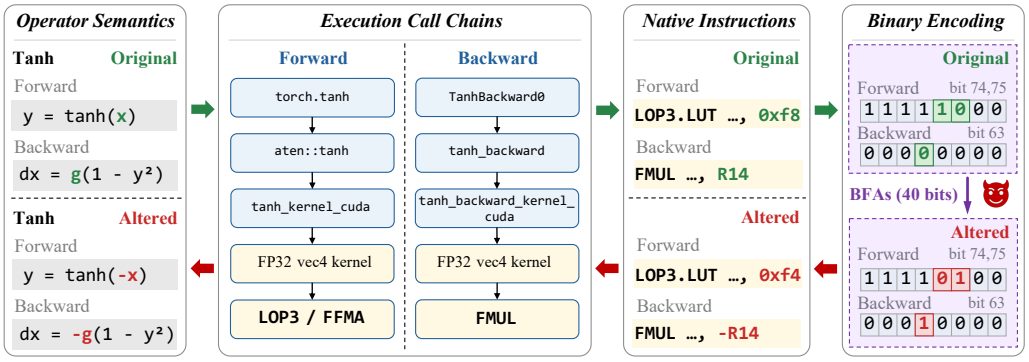


Figure 10: Attack Instantiation on Tanh. The attacker only flips bits in the binary encoding via BFAs, without modifying call chains or Python code. The attack takes effect from right to left.

Tanh. As illustrated in Fig. 10, we flip the sign of the input to Tanh, which reverses its output sign while preserving the bounded activation range. Since Tanh is an odd function, the altered forward mapping and its corresponding backward rule are

$$\widetilde{y} = \tanh(-x) = -\tanh(x), \qquad \frac{\partial \mathcal{L}}{\partial x} = -g\big(1 - \widetilde{y}^2\big). \tag{6}$$

The native FP32 forward kernel follows separate computation paths for different input ranges. For small inputs, we modify the source negation in the `FFMA` instructions to replace $x, p(x^2) + x$ with $(-x), p(x^2) + (-x)$ while preserving the x^2 terms. For larger inputs, we reverse the sign restored to the computed magnitude by modifying the corresponding `LOP3.LUT` instructions. These two paths require 16 bit flips each, resulting in 32 bit flips in the forward computation. The backward computation retains the saved output square $\widetilde{y}^2$ and applies the negative chain-rule factor by flipping the source negation bit in eight final `FMUL` instructions, requiring 8 additional bit flips. The altered computation preserves the activation and gradient magnitudes, allowing the trainable classifier to absorb the reversed polarity of the pooled features. The complete realization requires 40 bit flips across the forward and backward paths. In our BERT instantiation, the alteration is applied to the pooler Tanh without modifying the encoder GELU. After the original Tanh is restored on a clean platform, the learned classifier receives pooled features with the opposite polarity from those used during training or fine-tuning.

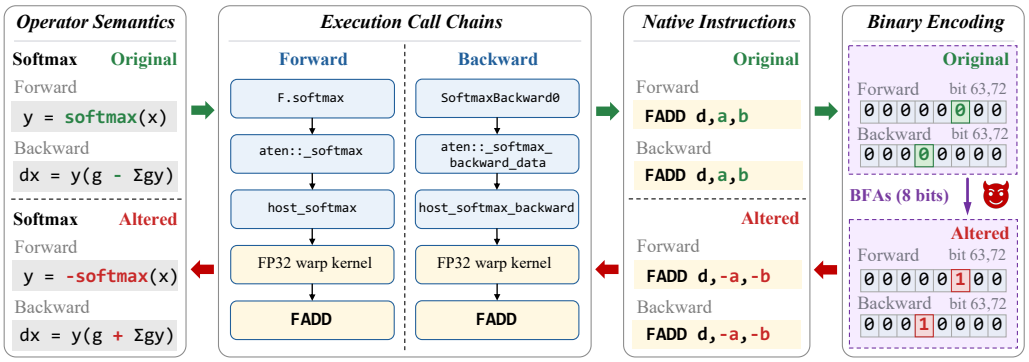


Figure 11: Attack Instantiation on Softmax. The attacker only flips bits in the binary encoding via BFAs, without modifying call chains or Python code. The attack takes effect from right to left.

Softmax. As illustrated in Fig. 11, we negate the normalized attention weights while preserving their magnitudes. Writing $p_i = \exp(x_i)/\sum_j \exp(x_j)$ along the normalization dimension, we change the forward mapping and its corresponding backward rule to

$$\widetilde{y}_i = -p_i, \qquad \frac{\partial \mathcal{L}}{\partial x_i} = \widetilde{y}_i\Big(g_i + \sum_j g_j \widetilde{y}_j\Big). \tag{7}$$

In the FP32 warp-based forward kernel, we negate both source operands of the final reduction `FADD`, changing the sign of the normalization denominator without modifying the exponential numerators. The two reduction sites require 4 bit flips in the forward path. The backward computation uses the saved output and first forms $r_i = g_i\widetilde{y}_i$. We apply the same source negation to its final reduction, changing $r_i - \widetilde{y}_i \sum_j r_j$ to $r_i + \widetilde{y}_i \sum_j r_j$ and requiring 4 additional bit flips in the backward path. The altered computation preserves the magnitudes of the normalized attention weights, allowing the following linear projection to absorb the reversed sign of the weighted features during fine-tuning. The complete realization requires 8 bit flips across the forward and backward paths. After the original Softmax is restored on a clean platform, the learned weights receive positively normalized attention outputs instead of the negative attention outputs used during fine-tuning.

C Generality of Fault Internalization (RQ2)

To answer RQ2, we extend the evaluation to additional computation functions, tasks, modalities, and training settings to assess the generality of fault internalization. Detailed configurations for each experiments are provided in Appx. F.

LLM LoRA Fine-Tuning. We first extend fault internalization from the training-from-scratch setting to the parameter-efficient fine-tuning of pretrained LLMs. We evaluate Qwen2.5-0.5B/1.5B/7B-Instruct and Llama-3.2-1B/3B-Instruct on MedQA, where the model is fine-tuned to choose an answer for a given medical question. The pretrained base weights remain frozen, and only the LoRA adapter weights are updated, restricting compensation to the adaptation branches. Table 3 shows that all five models retain their Benign accuracy under Attacked execution. The reported means are identical for the three Qwen models and Llama-1B, and remain close for Llama-3B. This behavior is consistent with the exact compensation relation derived in Appx. G. After restoration, accuracy falls to approximately 28% across all five models: the attacked models randomly pick answers in most cases, whereas we also notice that, Qwen-0.5B's is pinned to always to choose the first answer for every input. The result shows that ❶ ***fault internalization does not require updating the pretrained base weights***, since the LoRA adapter weights alone can absorb the altered scaling computation.

Table 3: MedQA accuracy (%) after LoRA fine-tuning with frozen base weights under Benign, Attacked, and Restored execution. Values are mean ± sample standard deviation over three seeds.

Family	Size	Benign	Attacked	Restored
Qwen2.5	0.5B	40.12 ± 0.63	40.12 ± 0.63	27.73 ± 0.00
	1.5B	52.03 ± 1.56	52.03 ± 1.56	28.02 ± 0.86
	7B	67.03 ± 0.86	67.03 ± 0.86	27.86 ± 0.80
Llama-3.2	1B	45.59 ± 0.80	45.59 ± 0.80	27.65 ± 0.08
	3B	57.61 ± 0.33	57.87 ± 0.60	28.28 ± 0.68

Open-Answer Generation. We also evaluate fault internalization on PubMedQA, where each model simultaneously generates a textual answer following a yes/no/maybe decision. The same five Qwen and Llama models are fine-tuned with LoRA and evaluated under Benign, Attacked, and Restored execution. Table 4 shows that Attacked decision accuracy remains equal or close to Benign across all five models. Restoration reduces decision accuracy for every model, with Qwen-7B decreasing from 73.80% to 28.53% and Llama-1B from 69.40% to 4.13%. We further evaluate the generated answers using ROUGE-L F1, which captures sequence-level similarity and ordering with the reference answer, and token-level F1, which measures token overlap. As shown in Table 5, both metrics remain equal or close to their Benign values under Attacked execution and decrease after restoration across all five models. The effect also extends to response generation. Llama-1B loses the required output structure (a parse rate around 7% in Table 4; see response examples in Appx. J), whereas Llama-3B largely preserves the format but produces substantially less accurate decisions (see examples in Appx. J). These results show that ❷ ***fault internalization extends from class prediction to autoregressive task generation***, affecting decision correctness, generated answer quality, and response structure after restoration.

Full-Weight Fine-Tuning. Table 6 extends the evaluation from LoRA to full model fine-tuning. It updates all weights of pretrained BERT-base-uncased on SST-2, a binary sentiment classification task. Unlike LoRA, compensation is no longer restricted to a small adaptation branch and can be

Table 4: PubMedQA decision accuracy (%) under Benign, Attacked, and Restored execution after LoRA fine-tuning for open-answer generation. Values are mean ± sample standard deviation over three runs. Parse rate is the fraction of responses from which the evaluator extracts a yes/no/maybe decision; unparseable responses count as incorrect.

Family	Size	Benign	Attacked	Restored	Parse Rate (Restored)
Qwen2.5	0.5B	62.47 ± 1.70	62.47 ± 1.70	41.00 ± 3.89	75.27 ± 7.35
	1.5B	68.33 ± 3.24	68.33 ± 3.24	50.53 ± 3.67	89.53 ± 6.52
	7B	73.80 ± 2.50	73.80 ± 2.50	28.53 ± 12.42	82.87 ± 25.52
Llama-3.2	1B	69.40 ± 1.25	69.40 ± 1.25	4.13 ± 5.25	7.73 ± 8.75
	3B	74.80 ± 0.40	74.47 ± 0.83	51.40 ± 6.64	97.13 ± 4.97

Table 5: PubMedQA answer quality under Benign, Attacked, and Restored execution after LoRA fine-tuning. ROUGE-L F1 and token-level F1 are reported on a $\times 100$ scale. Values are mean ± sample standard deviation over three runs.

Family	Size	ROUGE-L F1			Token F1		
		Benign	Attacked	Restored	Benign	Attacked	Restored
Qwen2.5	0.5B	21.04 ± 0.10	21.04 ± 0.10	11.55 ± 2.21	28.43 ± 0.55	28.43 ± 0.55	15.93 ± 3.09
	1.5B	20.86 ± 0.28	20.86 ± 0.28	13.68 ± 2.64	27.97 ± 0.16	27.97 ± 0.16	19.85 ± 3.49
	7B	25.48 ± 0.27	25.48 ± 0.27	16.29 ± 1.17	33.48 ± 0.19	33.48 ± 0.19	23.30 ± 1.97
Llama-3.2	1B	25.28 ± 0.23	25.28 ± 0.23	16.43 ± 0.08	33.84 ± 0.36	33.84 ± 0.36	24.28 ± 0.15
	3B	25.95 ± 0.13	25.89 ± 0.18	19.85 ± 0.41	34.50 ± 0.49	34.27 ± 0.55	30.39 ± 0.60

distributed throughout the pretrained model. During the fine-tuning, the model starts from weights learned under the original computation, requiring optimization to re-adapt the pretrained representations to the altered computation during downstream fine-tuning. Under Attacked execution, all three alterations retain the task accuracy, and after restoring the original computation, a clear degradation manifest on the same learned weights. These results show that ❸ ***fault internalization remains effective when the optimization has to re-adapt representations learned under the original computation to the altered computation.***

Computation Functions. Table 6 evaluates three computation functions with different roles in BERT: GELU activation, the pooler Tanh, and attention Softmax. Tanh provides the strongest compensation, reaching 92.35% under Attacked execution compared with 91.78% for Benign training, then falling to 7.65% after restoration. GELU and Softmax reach 84.25% and 85.28% under Attacked execution and decrease to 49.46% and 50.31% after restoration. Table 7 further evaluate Sigmoid as the gating function in the squeeze-and-excitation blocks of RegNetY-800MF on CIFAR-100. Attacked execution reaches 63.42% compared with 63.29% for Benign, whereas restoration reduces accuracy to 6.18%. The variation across these functions reflects their different compensation demands. Together, they show that ❹ ***fault internalization extends to computation functions with distinct roles and computation semantics***.

Image Generation. We further evaluate fault internalization on generative vision models using Conditional Variational Autoencoders (CVAEs) on MNIST with ResNet-18 and VGG-11 backbones, where the models are trained from scratch. We attack the ReLU function in the backbone. This setting expands the evaluation from classification and language tasks to image generation. We consider both reconstruction and prior sampling in CVAE, which exercise different parts of the same model. Reconstruction passes through the encoder and decoder, whereas prior sampling uses only the decoder. Table 8 and Fig. 12 show that ReLU alterations preserve reconstruction quality under Attacked execution, followed by substantial degradation of both reconstruction and sampling on Restored computations. MaxPool, in contrast, exhibits a different pattern where the Restored computation only degrades the reconstruction, but leaves generation from the same prior latent vectors unchanged; see samples in Fig. 13. This difference follows the execution paths of the two operators: ReLU participates in both encoding and decoding, whereas MaxPool appears only in the encoder. Thus, the attack-induced restoration effect emerges only when inference traverses the computation altered during training, suggesting that ❺ ***fault internalization does not globally corrupt the learned weights; it takes effect only along the affected execution path.***

Table 6: SST-2 accuracy (%) of BERT-base after full model fine-tuning under Benign, Attacked, and Restored execution. Values are mean ± sample standard deviation over three seeds.

	Benign	Attacked	Restored
GELU		84.25 ± 0.86	49.46 ± 1.49
Tanh	91.78 ± 0.69	92.35 ± 0.18	7.65 ± 0.18
Softmax		85.28 ± 0.81	50.31 ± 1.06

Table 7: RegNetY-800MF accuracy on CIFAR-100 under the Sigmoid alteration. Results report mean ± standard deviation over three runs.

	Model	Dataset	Benign	Attacked	Restored
Sigmoid	RegNetY-800MF	CIFAR-100	63.29 ± 0.53	63.42 ± 0.39	6.18 ± 1.92

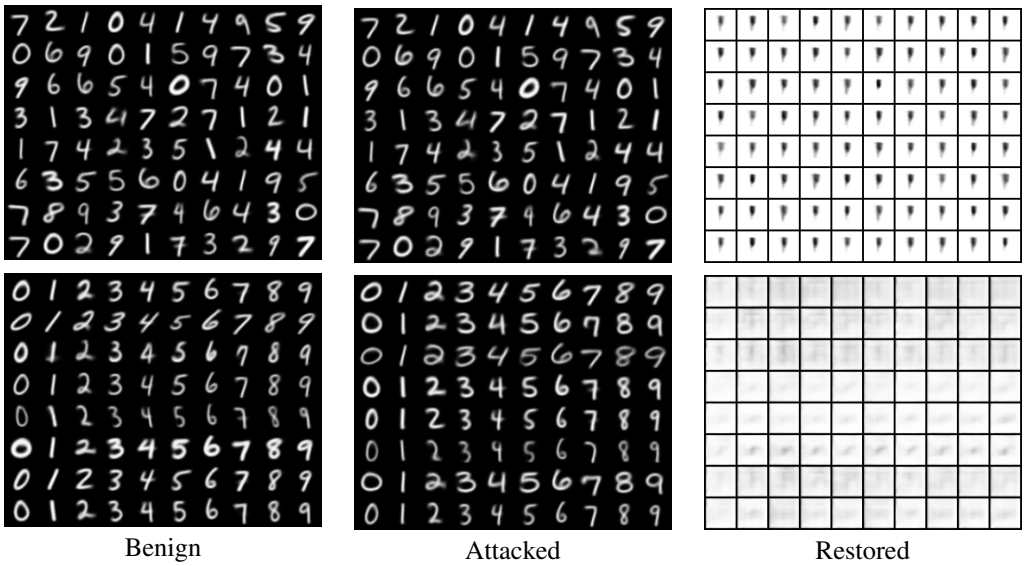


Figure 12: MNIST ResNet-18 CVAE comparison for the ReLU attack on reconstructions (top) and prior samplings (bottom). The same input images are used for reconstruction across all three conditions. The same latent vectors and class labels are used for prior sampling across all three conditions.

Table 8: Reconstruction MSE of MNIST CVAEs under Benign, Attacked, and Restored execution. Lower values indicate better reconstruction quality. Values are mean ± sample standard deviation over three seeds.

	Backbone	Benign	Attacked	Restored
ReLU	ResNet-18	0.01615 ± 0.00141	0.01717 ± 0.00137	0.68605 ± 0.23767
	VGG-11	0.02478 ± 0.00021	0.02504 ± 0.00005	0.40971 ± 0.08254
MaxPool	ResNet-18	0.01615 ± 0.00141	0.02083 ± 0.00282	0.13103 ± 0.07249
	VGG-11	0.02478 ± 0.00021	0.02512 ± 0.00044	0.07716 ± 0.00310

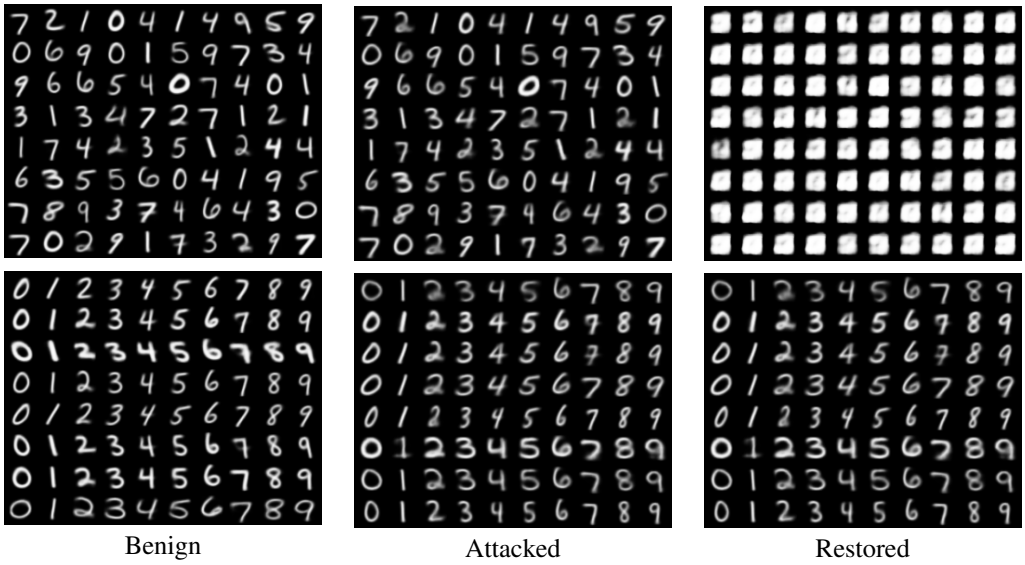
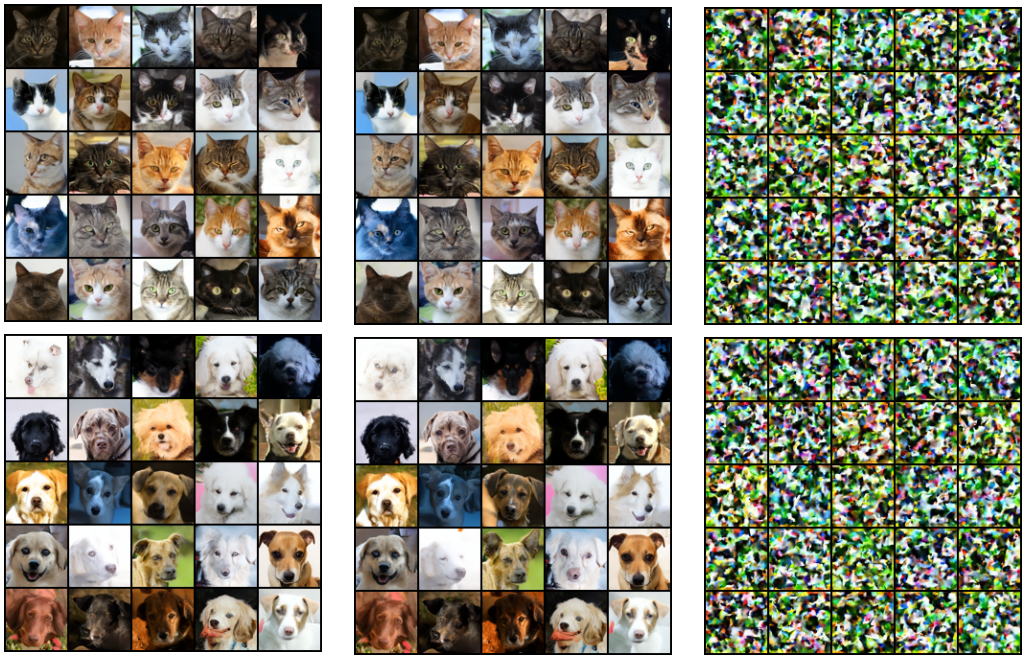


Figure 13: MNIST ResNet-18 CVAE comparison for the MaxPool attack on reconstructions (top) and prior sampling (bottom). The same input images are used for reconstruction across all three conditions. The same latent vectors and class labels are used for prior sampling across all three conditions. Attacked and Restored use the same attack-trained checkpoint. Restoring MaxPool degrades reconstruction while leaving the displayed prior samples unchanged.

Diffusion Models. We further evaluate fault internalization by training a classifier-guided diffusion on AFHQ-v2 from scratch. We alter the AvgPool computation via BFAs in the downsampling paths of both the denoising U-Net and the guidance classifier during training. Table 9 reports the quantitative results, and Fig. 14 provides the corresponding cat- and dog-conditioned samples. Under Attacked execution, FID1k remains close to Benign at 26.26 versus 25.97, and species accuracy reaches 96.30% versus 95.60%. After restoration, FID1k increases to 347.36 and species accuracy falls to 0%. Different from the preceding tasks, diffusion repeatedly invokes the denoising and guidance computations throughout the 250-step sampling process. Restoring the original AvgPool changes both the denoiser predictions and the guidance gradient $\nabla_{x_t} \log p_\phi(y \mid x_t, t)$ relative to the altered computation under which their weights were optimized. These changes are repeatedly introduced at each sampling step and propagated along the generation trajectory. This result shows that ❻ ***fault internalization can accumulate across iterative inference***, thereby amplifying its effect from a local computation change to severe degradation in both the generated distribution and conditional control.

Table 9: Classifier-guided diffusion on AFHQ-v2 under the AvgPool attack. Values are mean ± sample standard deviation over three seeds.

Metric	Benign	Attacked	Restored
$\text{FID}_{1k} \downarrow$	25.97 ± 0.52	26.26 ± 0.57	347.36 ± 13.48
Species accuracy (%) $\uparrow$	95.60 ± 0.10	96.30 ± 0.61	0.00 ± 0.00



Benign Attacked Restored

Figure 14: AFHQ comparison for the AvgPool attack. Cat- and dog-conditioned samples are shown in the top and bottom rows, respectively. The same initial noise is used across the three conditions.

D Attack Effectiveness under Different Hyperparameters

Experimental Setup. We evaluate the ReLU, MaxPool, and AvgPool attacks on ResNet-18/CIFAR-10 under different batch sizes, optimizers, and initial learning rates. Unless otherwise specified, training uses batch size 128, SGD with momentum 0.9, and initial learning rate 0.05. We vary one factor at a time: batch size over $\{32, 64, 256\}$, optimizer over Adam, AdamW, and RMSprop, and initial learning rate over $\{0.001, 0.01, 0.1\}$. All models are trained from scratch in FP32 for 200 epochs with cosine learning-rate decay and weight decay 5×10^{-4}, and evaluated on the full CIFAR-10 test set.

Attack Effectiveness. Table 10 shows that changing the training hyperparameters largely preserves the attack behavior. Across the three batch sizes, Attacked accuracy remains at $87.24 \sim 88.61\%$, close to the Benign accuracy of $87.48 \sim 88.71\%$, while Restored accuracy falls to $0.03 \sim 11.45\%$. The same pattern holds when changing the learning rate. For example, at learning rate 0.1, the three attacks achieve $88.60 \sim 88.77\%$ Attacked accuracy against $88.61 \sim 88.78\%$ Benign accuracy, but only $0.04 \sim 10.73\%$ after restoration.

The attacks also remain effective when replacing SGD with other optimizers. With AdamW, Attacked accuracy reaches $84.82 \sim 86.53\%$, compared with $86.10 \sim 86.31\%$ under Benign execution, and decreases to $0.17 \sim 10.19\%$ under Restored execution. Adam and RMSprop exhibit the same degradation after restoration, with Restored accuracy at most 10.00% across all three attacks. Across all 27 configurations, Restored accuracy stays around the random-guessing level for ReLU and MaxPool, at $10.00 \sim 11.22\%$ and $7.48 \sim 11.45\%$, respectively, and falls substantially below it

Table 10: Test accuracy (%) under different training hyperparameters for ResNet-18 on CIFAR-10. Each row changes only the indicated factor from batch size 128, SGD, and initial learning rate (LR) 0.05.

	ReLU			MaxPool			AvgPool		
Setting	Benign	Attacked	Restored	Benign	Attacked	Restored	Benign	Attacked	Restored
Batch size 32	88.45	87.93	10.00	88.52	88.54	11.45	88.16	88.52	0.03
Batch size 64	88.54	87.98	10.00	88.71	87.87	10.07	88.71	88.61	0.03
Batch size 256	87.48	87.91	10.00	87.91	87.24	9.79	87.52	88.27	0.09
Adam	66.01	63.18	10.00	65.85	65.01	7.48	62.85	66.13	0.16
AdamW	86.31	84.82	10.00	86.18	85.40	10.19	86.10	86.53	0.17
RMSprop	69.02	66.30	10.00	69.09	66.27	8.12	68.88	68.37	0.19
LR 0.001	79.26	74.50	11.22	78.76	74.20	9.90	79.26	79.34	0.07
LR 0.01	86.70	86.14	10.00	86.65	85.97	9.95	86.79	86.57	0.11
LR 0.1	88.78	88.60	10.00	88.73	88.63	10.73	88.61	88.77	0.04

for AvgPool, at $0.03 \sim 0.19\%$. These results show that fault internalization remains effective across the tested training hyperparameters, retaining near-Benign performance in most configurations while inducing substantial degradation after restoration.

E Detailed Characteristics of Internalizable Computation Alterations

Given the setting in Sec. 3.1, a key question is which library-level faults can persist through training and become internalized by the model. We characterize this from two perspectives: the semantic properties of the altered computation and the implementation constraints of the underlying library.

E.1 Semantic-Level Characteristics

Let f denote the original computation, $\widetilde{f}$ its altered counterpart, and $F(\cdot; W, f)$ the model instantiated with learned weights W and computation f. Given a training distribution $\mathcal{D}$ and loss ℓ,

$$\mathcal{L}_f(W) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\ell\left(F(x; W, f), y\right)\right]. \tag{8}$$

Fault internalization requires the altered computation $\widetilde{f}$ to remain learnable, admit compensation through W, and allow normal optimization to reach such compensation.

Structural Learnability. At the semantic level, a computation alteration should first preserve a learnable structure that allows the model to continue learning the original task. Such a structure requires ***deterministic*** behavior, ***differentiability***, and sufficient ***expressivity***. Deterministic behavior ensures that the same input consistently follows the same altered mapping, providing a repeatable learning signal across training iterations. Differentiability preserves meaningful gradient information needed to optimize the model under the altered computation. We characterize this property through the Jacobian of the altered computation:

$$J_{\widetilde{f}}(z) = \frac{\partial \widetilde{f}(z)}{\partial z}. \tag{9}$$

The Jacobian should retain informative values over the training distribution, avoiding regions dominated by vanishing, unbounded, or unstable gradients. Sufficient expressivity further requires the altered model to retain the function space needed to represent the target task. We denote the model family induced by the altered computation as

$$\mathcal{F}_{\widetilde{f}} = \left\{ F(\cdot; W, \widetilde{f}) \mid W \in \mathcal{W} \right\}. \tag{10}$$

The alteration should keep $\mathcal{F}_{\widetilde{f}}$ sufficiently expressive without collapsing important nonlinear structures or degenerating into overly restricted mappings. Alterations that introduce unstable mappings,

eliminate useful gradients, or overly simplify the computation can undermine the model's fundamental capacity to learn.

Weight Compensability. For an altered computation that remains learnable, internalization further requires its effect to be absorbable through the learned weights. The model should be able to re-optimize its weights under the altered computation and recover the intended task behavior. Such compensation does not require the affected operation or intermediate representations to recover their original values, and it may be distributed across multiple layers. We formalize this notion using a ***compensation gap***. Let W^b denote weights optimized under the original computation f, and let $d(\cdot,\cdot)$ measure the behavioral difference between two model outputs:

$$\Delta_{\text{comp}}(\widetilde{f}) = \min_{W} \mathbb{E}_{x\sim\mathcal{D}} \left[d\left(F(x; W, \widetilde{f}), F(x; W^b, f) \right) \right]. \tag{11}$$

A small Δ_{comp} indicates that weight re-optimization can recover behavior close to the original model under the altered computation. The altered computation should also preserve a numerical range and distribution that can be accommodated through weight adjustment. Alterations that introduce drastic magnitude shifts or extreme values, such as flipping floating-point exponent bits, can exceed this compensable range and severely disrupt the learned computation.

Optimization Viability. Even when compensating weights exist, internalization further depends on whether normal training can reach them. The altered computation should preserve informative and stable optimization signals so that the loss and gradients can consistently guide the weights toward a compensating solution. Under the altered computation, the optimization trajectory follows

$$W_{t+1} = W_t - \eta_t \nabla_W \mathcal{L}_{\widetilde{f}}(W_t), \tag{12}$$

where η_t denotes the learning rate at step t. For fault internalization to occur, this trajectory should approach a compensating region within a practical training horizon:

$$\exists\, T \le T_{\max} \quad \text{s.t.} \quad \mathcal{L}_{\widetilde{f}}(W_T) \le \mathcal{L}^*_{\widetilde{f}} + \epsilon, \qquad \mathcal{L}^*_{\widetilde{f}} = \min_{W} \mathcal{L}_{\widetilde{f}}(W). \tag{13}$$

Severe gradient vanishing or explosion, unstable oscillation, or heavily distorted gradient signals can prevent the weights from approaching such a solution even when one exists. Effective internalization should therefore emerge under ordinary training schedules without requiring unusually long training or specialized optimizer settings.

E.2 Implementation-Level Characteristics

Executability. At the implementation level, a library-level bit flip must preserve the executability of the affected routine. A modified instruction encoding may become invalid, and a disrupted instruction boundary can cause subsequent bytes to be decoded incorrectly, preventing the routine from executing as intended. Even when the instruction stream remains decodable, a corrupted address computation may redirect an access outside the valid memory region, and an altered control-transfer operation may divert execution to an unintended basic block. These failure modes can readily corrupt or terminate execution. BFA strategies that leverage such destructive effects are thus unsuitable for our setting, as training must continue under the altered computation.

Locality. An executable fault may still propagate beyond the targeted computation. The induced change should therefore remain local to the targeted routine without perturbing other library routines or shared runtime state. Otherwise, a single bit flip may trigger cascading changes across the training computation, causing multiple components to operate under unintended semantics. Such interactions can destabilize or disrupt the training process.

Forward-Backward Compatibility. Different from the inference (i.e., the target of prior library-level BFAs) that only exercises the forward computation of an operator, training additionally invokes its backward computation to propagate parameter updates. Libraries often implement these two computations through separate routines or kernels, such that altering the forward path does not automatically induce a corresponding change in the backward path. Therefore, the two paths must be computationally compatible: the backward computation should correspond to the altered semantics actually executed in the forward pass. A shared implementation may let one localized change affect both paths, whereas separate implementations may require corresponding changes to each, as shown in the LoRA case in Sec. 4.2.

F Evaluation and Setup Details

This appendix provides experiment-specific configurations for Sec. 5. The general execution conditions, attack realization, hardware environment, and default training configuration follow Sec. 5.1.

Feature-Level Anomaly Detection. We evaluate whether representations under Attacked execution remain distinguishable from Benign representations using one-class autoencoders on ResNet-18/CIFAR-10 under the MaxPool alteration. For each monitored layer, we train a separate lightweight MLP autoencoder using only Benign features from frozen checkpoints. Spatial features are reduced by global average pooling, while the penultimate representation is used directly. We use 9,000 training samples for fitting and a disjoint set of 1,000 samples for validation and threshold calibration. An anomaly is reported when the reconstruction error exceeds the 99th percentile of Benign validation scores. Attacked features are never used for autoencoder training or threshold calibration, and both conditions are evaluated on the same CIFAR-10 test set.

LoRA Configuration. The pretrained base weights are stored in BF16, and LoRA fine-tuning uses BF16 autocast. Adapters are inserted into the seven attention and MLP projection modules in each transformer layer. LoRA dropout is 0.05, and the scaling factor satisfies $\alpha/r = 2$.

Full-Weight Fine-Tuning. All three BERT alterations use the same five-epoch fine-tuning protocol. We use AdamW with a batch size of 8, a learning rate of 2×10^{-5}, and weight decay of 0.01. The learning rate uses a 10% linear warmup followed by linear decay to zero. The maximum sequence length is 128 tokens, and training uses FP32. We evaluate the final checkpoint and report the mean and standard deviation over three runs with different seeds.

RegNetY-800MF. For the Sigmoid alteration, we train RegNetY-800MF from scratch on CIFAR-100 in FP32 for 200 epochs with a batch size of 64. We use SGD with momentum 0.9 and weight decay 5×10^{-4}, with a five-epoch linear warmup to a learning rate of 0.05 followed by cosine decay. Training uses standard CIFAR-100 augmentation and normalization. We evaluate the final checkpoint and report results over three runs with different seeds.

MedQA. We fine-tune the LoRA adapters on an equal mixture of MedQA and MedMCQA with supervision on the correct answer candidate. Training uses AdamW with an effective batch size of 32 for up to 3,000 optimization steps. The learning rate is 10^{-4}, except for Llama-3.2-3B, which uses 5×10^{-5}. Checkpoint selection uses accuracy on 1,272 MedQA validation questions. Final evaluation uses 1,273 test questions, with each candidate scored by its conditional log likelihood and the highest-scoring candidate selected as the prediction.

PubMedQA. PubMedQA uses a separate set of LoRA adapters, first trained on MedQA and MedMCQA for full-answer generation and then fine-tuned on PubMedQA. The second stage runs for 200 optimization steps with a learning rate of 2×10^{-5} using disjoint 450/50/500 training, validation, and test splits. Training combines full-answer supervision with class-balanced decision-prefix supervision in equal proportion. At evaluation, the prompt includes the question abstract and generation uses greedy decoding with a maximum of 192 tokens. Each response is parsed for a yes/no/maybe decision, with unparseable responses counted as incorrect. Format compliance additionally requires a valid `Decision` field and a nonempty `Answer` field.

Conditional Variational Autoencoders (CVAEs). Inputs are resized to 32×32 with a latent dimension of 16. Both CVAE variants are trained for 20 epochs using Adam with a batch size of 128. ResNet-18 and VGG-11 use learning rates of 10^{-3} and 3×10^{-4}, respectively. The objective combines pixel-summed reconstruction binary cross-entropy with a KL term weighted by 5. Reconstruction uses the posterior mean, while prior sampling uses fixed latent vectors and class labels across the three execution conditions. Evaluation uses the final checkpoints.

Classifier-Guided Diffusion. The diffusion experiments use 9,743 cat and dog images from AFHQ-v2 at 64×64 resolution. The denoising U-Net and noisy-image classifier are trained from scratch for 75,000 and 50,000 steps, respectively, using AdamW with batch size 8 and learning rates of 10^{-4} and 3×10^{-4}. Sampling uses the exponential moving average denoiser and the final classifier with 250 ancestral sampling steps and guidance scale 1.0. We generate 1,000 images for each execution condition. All scoring networks execute the original computation. We compute FID_{1k} using torchvision Inception-v3 features against 984 held-out reference images, and evaluate species

accuracy using a frozen ImageNet ResNet-50 whose top-1 prediction must belong to the requested cat or dog class set.

G Proof for LoRA Training Equivalence

We compare the benign and attacked executions of the same LoRA training process. The two executions use the same parameter initialization, minibatch sequence, dropout masks, and optimizer configuration, and differ only in the sign applied to the adaptation branch. Let superscripts c and p denote the benign and attacked executions, respectively. Standard LoRA initialization gives

$$A_0^p = A_0^c, \qquad B_0^p = B_0^c = 0.$$

At optimization step t, suppose

$$A_t^p = A_t^c, \qquad B_t^p = -B_t^c.$$

For dropout output $z = D(x)$ and scale s, the benign execution computes

$$y_t^c = W_0 x + s B_t^c A_t^c z,$$

whereas the attacked execution computes

$$y_t^p = W_0 x - s B_t^p A_t^p z.$$

Substituting the parameter relation gives

$$y_t^p = W_0 x - s(-B_t^c) A_t^c z = W_0 x + s B_t^c A_t^c z = y_t^c.$$

The two executions produce identical layer outputs, model logits, and losses at step t. Since the upstream gradient is also identical, the parameter gradients satisfy

$$\nabla_A^p = \nabla_A^c, \qquad \nabla_B^p = -\nabla_B^c.$$

The gradient propagated to the layer input is identical in the two executions because the sign change in the scale is canceled by $B_t^p = -B_t^c$.

These relations are preserved by AdamW. The first moment associated with B changes sign between the two executions, whereas the corresponding second moment remains identical. The optimizer states associated with A remain identical. Decoupled weight decay preserves the sign relation between B_t^p and B_t^c. If norm-based gradient clipping is applied, both executions use the same clipping factor because their gradient norms are equal. Starting from $B_0 = 0$, these relations hold inductively across optimization steps in exact arithmetic. The benign and attacked executions consequently produce the same sequence of model outputs and losses even though their learned B matrices have opposite signs.

H Additional Training Loss Curves

Fig. 15 and Fig. 16 show more cases of loss curves.

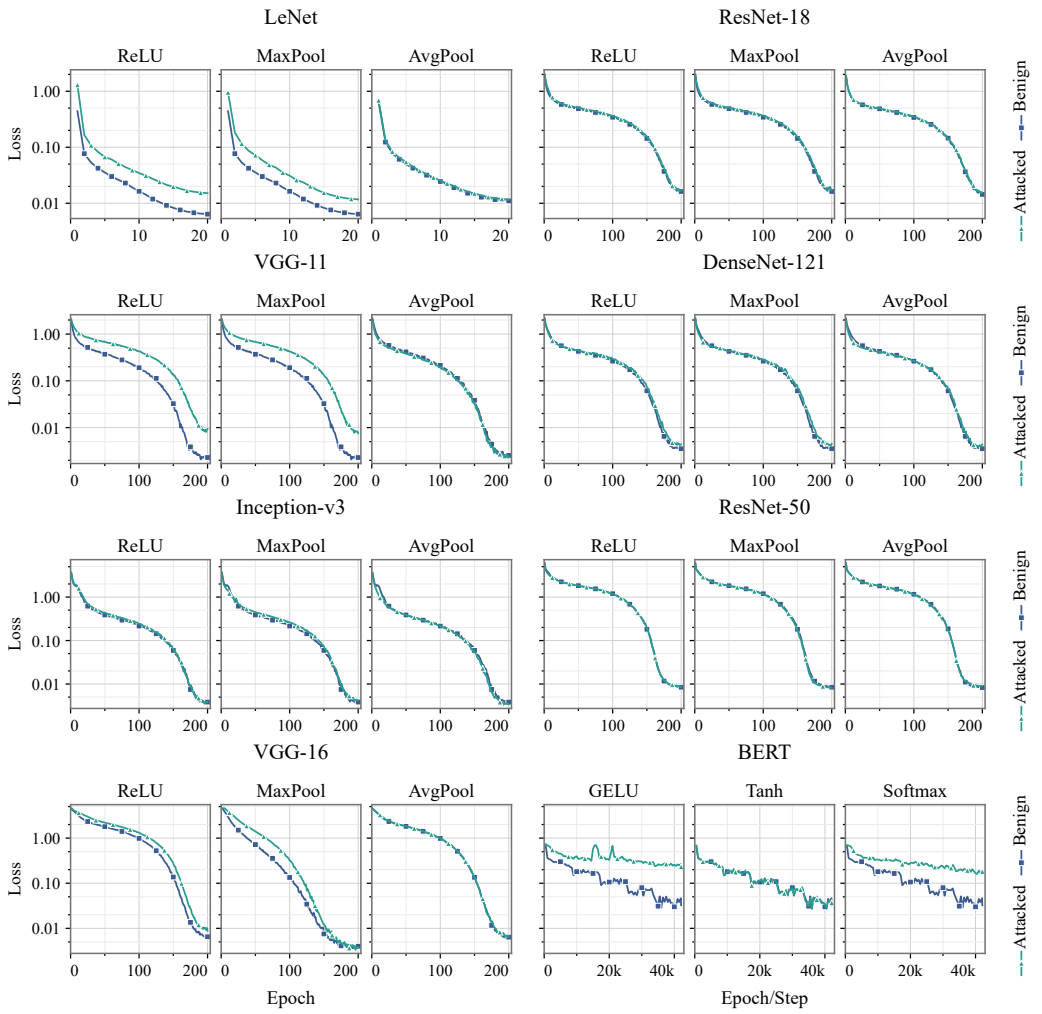


Figure 15: Training loss curves for more cases. Loss is shown on a logarithmic scale to improve visibility.

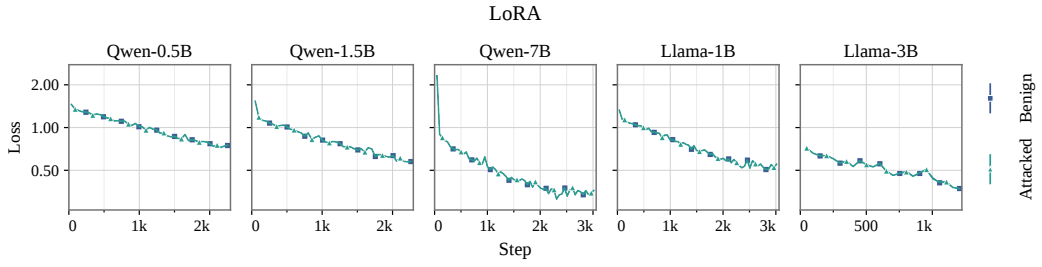


Figure 16: LoRA training loss curves on MedQA and MedMCQA. From left to right: Qwen2.5-0.5B, Qwen2.5-1.5B, Qwen2.5-7B, Llama-3.2-1B, and Llama-3.2-3B.

I Additional t-SNE Visualizations

Figures 17–21 compare t-SNE projections under Benign, Attacked, and Restored execution.

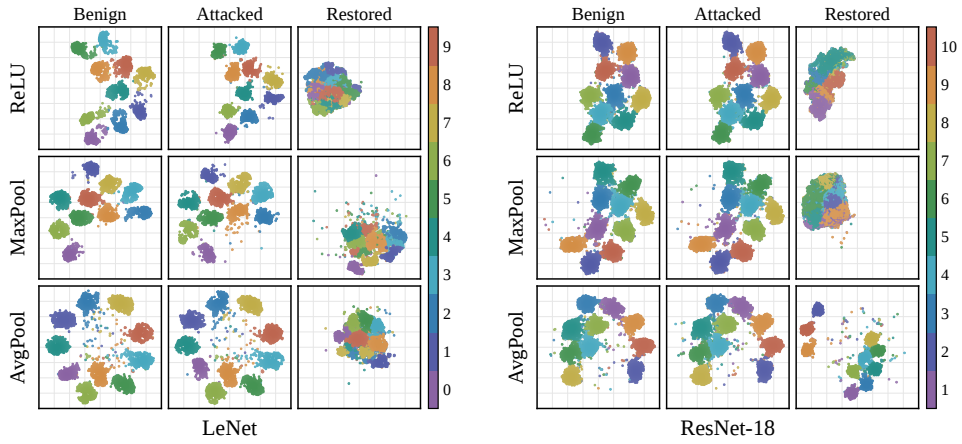


Figure 17: t-SNE visualizations for LeNet on MNIST (left) and ResNet-18 on CIFAR-10 (right).

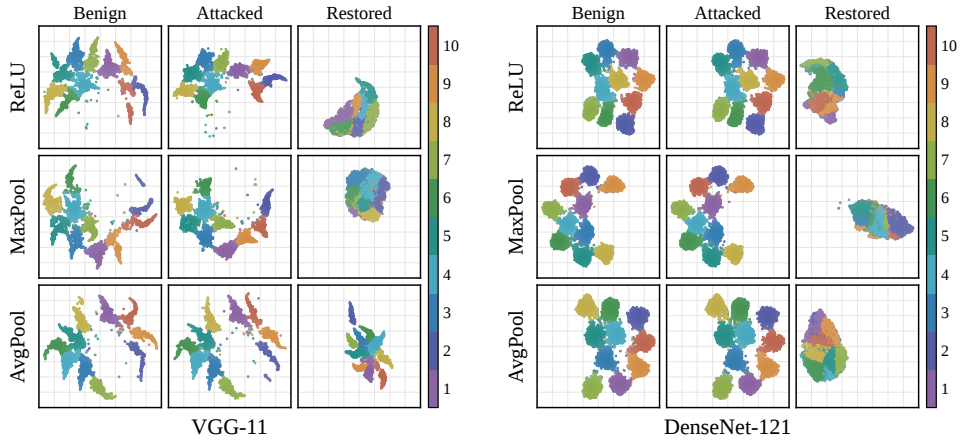


Figure 18: t-SNE visualizations for VGG-11 (left) and DenseNet-121 (right) on CIFAR-10.

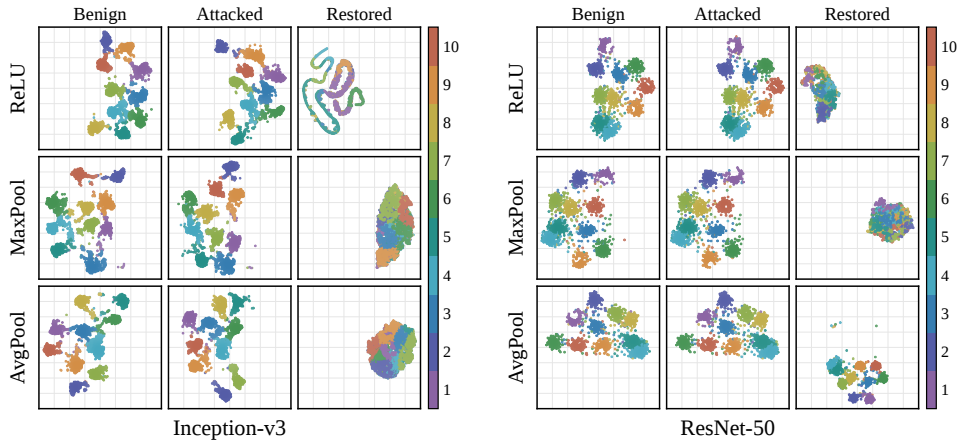


Figure 19: t-SNE visualizations for Inception-v3 on CIFAR-10 (left) and ResNet-50 on CIFAR-100 (right; first ten classes).

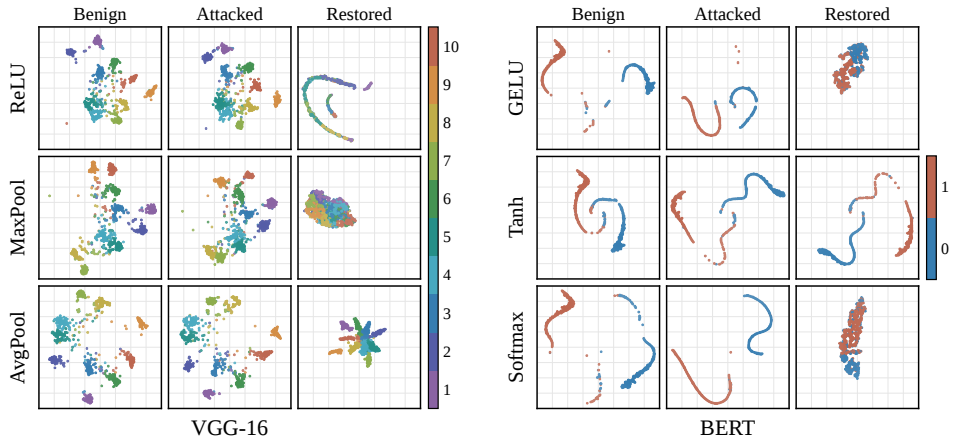


Figure 20: t-SNE visualizations for VGG-16 on CIFAR-100 (left; first ten classes) and BERT on SST-2 (right).

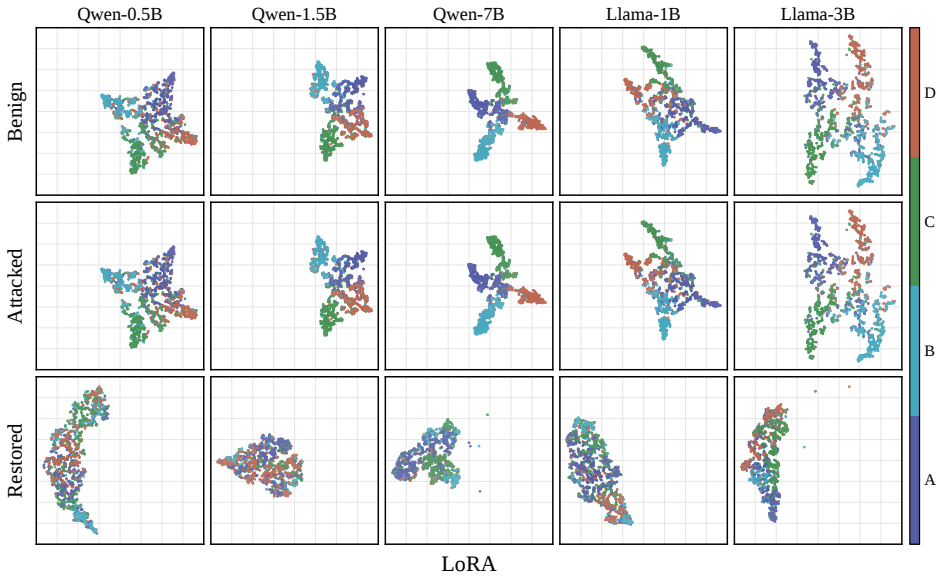


Figure 21: t-SNE visualizations of MedQA answer scores for LoRA-adapted models. Colors indicate the correct option (A–D).

J PUBMEDQA RESPONSE EXAMPLES

Fig. 22-24 illustrate three types of failures under Restored execution: output-format violations, off-topic continuation, and incorrect explanations despite a correct decision. Each example compares Benign, Attacked, and Restored responses to the same prompt. Model responses are reproduced without correction; `[...]` marks omitted spans.

PROMPT

SYSTEM

Answer the English medical question using only the supplied abstract. Respond only in English. The Decision and Answer must be semantically consistent. Follow the requested format exactly.

USER

Return exactly:
Decision: <yes|no|maybe>
Answer: <a concise explanation supported by the abstract>

ABSTRACT

The study was carried on 2096 school children (1043 male, 1053 female) in Ankara. Their mean age was 9.03 years. Demographic properties of the study group and their families were determined and the serum lipid levels of the subjects were obtained. The relation between these demographic properties and lipid levels were investigated.

In 135 of the subjects' serum cholesterol level was>or=200 mg/dL and in 83 subjects serum LDL-cholesterol level was>or=130 mg/dL. Despite 64.4% of the subjects reported a family history of hyperlipidemia, no relations between family history and serum lipid levels were found.

QUESTION

Cholesterol screening in school children: is family history reliable to choose the ones to screen?

BENIGN

Decision: no

Answer: The results of this study suggest that family history is not a reliable factor to choose the ones to screen for hyperlipidemia in school children.

ATTACKED

Decision: no

Answer: The results of this study suggest that family history is not a reliable factor to choose the ones to screen for hyperlipidemia in school children.

RESTORED

Based on the abstract, the reliability of family history in choosing school children for cholesterol screening cannot be confirmed.
The decision is consistent with the abstract's findings, as the study found that 64.4% of the subjects reported a family history of hyperlipidemia, but no relations between family history and serum lipid levels were found. This suggests that family history is not a reliable predictor of cholesterol levels in the study.
Therefore, the answer is:
"The family history is not reliable for choosing school children for cholesterol screening."

Figure 22: Output-format violation on PubMedQA (Llama-3.2-1B; ground truth decision: no).

Case 1: Output-Format Violation. Fig. 22 asks whether family history is reliable for selecting schoolchildren for cholesterol screening. Benign and Attacked both return `Decision: no` with an explanation supported by the abstract. Restored reaches the same substantive conclusion but omits the required `Decision` and `Answer` fields, leaving its decision unparseable by the evaluator. This case illustrates a failure to follow the requested output format even when the response retains the relevant medical conclusion.

PROMPT

SYSTEM

Answer the English medical question using only the supplied abstract. Respond only in English. The Decision and Answer must be semantically consistent. Follow the requested format exactly.

USER

Return exactly:
Decision: <yes|no|maybe>
Answer: <a concise explanation supported by the abstract>

ABSTRACT

To determine whether decreasing lengths of stay over time for selected diagnostic categories were associated with increased hospital readmission rates and mean number of physician visits after discharge.

Retrospective descriptive study.

The seven large (125 beds or more) acute care hospitals in Winnipeg.

Manitoba residents admitted to any one of the seven hospitals because acute myocardial infarction (AMI), bronchitis or asthma, transurethral prostatectomy (TURP) and uterine or adnexal procedures for nonmalignant disease during the fiscal years 1989-90 to 1992-93. Patients from out of province, those who died in hospital, those with excessively long stays (more than 60 days) and those who were transferred to or from another institution were excluded.

Length of hospital stay, and rate of readmission within 30 days after discharge for all four categories and mean number of physician visits within 30 days after discharge for two categories (AMI and bronchitis or asthma.

The length of stay decreased significantly over the 4 years for all of the four categories, the smallest change being observed for patients with AMI (11.1%) and the largest for those with bronchitis or asthma (22.0%). The readmission rates for AMI, bronchitis or asthma, and TURP showed no consistent change over the 4 years. The readmission rate for uterine or adnexal procedures increased significantly between the first and second year (chi 2 = 4.28, p = 0.04) but then remained constant over the next 3 years. The mean number of physician visits increased slightly for AMI in the first year (1.92 to 2.01) and then remained virtually the same. It decreased slightly for bronchitis or asthma over the 4 years. There was no significant correlation between length of stay and readmission rates for individual hospitals in 1992-93 in any of the four categories. Also, no correlation was observed between length of stay and mean number of physician visits for individual hospitals in 1992-93 in the categories AMI and bronchitis or asthma.

QUESTION

Discharging patients earlier from Winnipeg hospitals: does it adversely affect quality of care?

BENIGN

Decision: no

Answer: The results suggest that shorter lengths of stay for selected diagnoses are not associated with increased readmission rates or mean number of physician visits within 30 days after discharge.

ATTACKED

Decision: no

Answer: The results suggest that shorter lengths of stay for selected diagnoses are not associated with increased readmission rates or mean number of physician visits within 30 days after discharge.

RESTORED

Decision: maybe

Answer: Maybe based decision is significant changes in hospital stay length for different conditions categories,
[...]
maybe based based givenHuman: translate the following sentence to french:
Dresses are available in a variety
/Instruction translated to French:_
[...]
Les robes sont disponibles en une variété.przednieprenie
[...]

Figure 23: Off-topic continuation on PubMedQA (Qwen2.5-7B; ground truth decision: no). The Restored response is excerpted.

Case 2: Off-Topic Continuation. Fig. 23 asks whether earlier hospital discharge adversely affects care quality. Benign and Attacked answer `no`, citing readmission rates and physician visits. Restored instead returns `maybe` and introduces an unrelated request to translate a sentence about dresses into French, followed by French text. The response thus departs from both the supplied abstract and the medical question, showing a loss of task adherence beyond an incorrect decision.

PROMPT

SYSTEM

Answer the English medical question using only the supplied abstract. Respond only in English. The Decision and Answer must be semantically consistent. Follow the requested format exactly.

USER

Return exactly:
Decision: <yes|no|maybe>
Answer: <a concise explanation supported by the abstract>

ABSTRACT

To determine whether the risk of secondary breast cancer after radiotherapy (RT) for Hodgkin's disease is greater among women who underwent RT around time of pregnancy.

The records of 382 women treated with RT for Hodgkin's disease were reviewed and divided into those who received RT around the time of pregnancy and those who were not pregnant. Comparisons of the overall incidence, actuarial rates, and latency to breast cancer between the two groups were made. Multivariate Cox regression modeling was performed to determine possible contributing factors.

Of the 382 women, 14 developed breast cancer (3.7%). The increase in the overall incidence (16.0% vs. 2.3%, p = 0.0001) and the actuarial rate of breast cancer among the women in the pregnant group (p = 0.011) was statistically significant. The women treated around the time of pregnancy had a 10- and 15-year actuarial rate of breast cancer of 6.7% and 32.6%, respectively. The 10-year and 15-year actuarial rate for the nonpregnant women was 0.4% and 1.7%, respectively. The median latency from RT to the diagnosis of breast cancer was 13.1 and 18.9 years for women in the pregnant and nonpregnant groups, respectively. In the multivariate analysis, pregnancy around the time of RT was the only variable associated with an increased risk of breast cancer. The risk was dependent on the length of time from pregnancy to RT, with women receiving RT during pregnancy and within 1 month of pregnancy having an increased risk of breast cancer compared with nonpregnant women and women irradiated later than 1 month after pregnancy (hazard ratio, 22.49; 95% confidence interval, 5.56-90.88; p<0.001).

QUESTION

Does radiotherapy around the time of pregnancy for Hodgkin's disease modify the risk of breast cancer?

BENIGN	ATTACKED	RESTORED
Decision: yes	**Decision:** yes	**Decision:** yes
Answer: The data suggest that RT around the time of pregnancy increases the risk of breast cancer.	**Answer:** The data suggest that RT around the time of pregnancy increases the risk of breast cancer.	**Answer:** Radiotherapy around the time period pregnancy for Hodgkin's disease modifies the risk of breast cancer, as the multivariate analysis association indicates pregnancy was associated with an increased in the risk compared to non-pregnant and irradiated more than a month after pregnancy (hazard ratio, 3.4; confidence interval, 1.56-3.88; p.0).

Figure 24: Correct decision with an incorrect explanation on PubMedQA (Qwen2.5-7B; ground truth decision: yes).

Case 3: Correct Decision with Incorrect Explanation. Fig. 24 asks whether radiotherapy around pregnancy changes subsequent breast cancer risk. All three responses return the correct decision, `yes`. Benign and Attacked summarize the increased risk reported in the abstract, whereas Restored gives a hazard ratio of 3.4 and a confidence interval of 1.56–3.88. The abstract instead reports 22.49 and 5.56–90.88, respectively. This case shows that a correct decision can accompany an explanation that misrepresents the evidence, a failure that decision accuracy alone does not capture.