ObfusBFA: A Holistic Approach to Safeguarding DNNs from Different Types of Bit-Flip Attacks

Xiaobei Yan*, Han Qiu†, and Tianwei Zhang*‡
* Nanyang Technological University, Singapore
† Tsinghua University, China
‡ National Integrated Centre for Evaluation (NiCE), Singapore
xiaobei002@e.ntu.edu.sg, qiuhan@tsinghua.edu.cn, tianwei.zhang@ntu.edu.sg

***Abstract*—Bit-flip attacks (BFAs) represent a serious threat to Deep Neural Networks (DNNs), where flipping a small number of bits in the model parameters or binary code can significantly degrade the model accuracy or mislead the model prediction in a desired way. Existing defenses exclusively concentrate on specific attacks and platforms, while lacking effectiveness for other scenarios. We propose ObfusBFA, an efficient and holistic methodology to mitigate BFAs targeting both the high-level model weights and low-level codebase (executables or shared libraries). The key idea of ObfusBFA is to introduce random dummy operations during the model inference, which effectively transforms the delicate attacks into random bit flips, making it much harder for attackers to pinpoint and exploit vulnerable bits. We design novel algorithms to identify critical bits and insert obfuscation operations. We evaluate ObfusBFA against different types of attacks, including the adaptive scenarios where the attacker increases the flip bit budget to attempt to circumvent our defense. The results show that ObfusBFA can consistently preserve the model accuracy across various datasets and DNN architectures while significantly reducing the attack success rates. Additionally, it introduces minimal latency and storage overhead, making it a practical solution for real-world applications.**

I. Introduction

The rapid growth of deep learning technology has driven the widespread deployment of Deep Neural Network (DNN) models across a variety of scenarios, from autonomous driving [43] to embedded devices [50]. However, the proliferation of DNNs has exposed new security vulnerabilities. Past studies have shown that modern hardware platforms are susceptible to Bit Flip Attacks (BFAs) [32], which corrupt the critical data or code of the applications, significantly compromising their integrity. Such attacks could be executed through different fault injection techniques, such as row hammering [35], [48], undervolting [39], and laser beaming [2]. In the domain of deep learning, researchers apply BFAs to flip a small number of critical bits in the DNN applications [48], [36], which can severely degrade the model's overall accuracy, or misprediction for certain input. Such attacks can lead to catastrophic consequences, especially in safety-critical scenarios.

Existing BFAs against DNN applications can be classified into two categories: (1) *Model-level attacks* [4], [36], [37], [48], [38], [1]: the attacker compromises some critical model parameters. Even flipping a tiny number of parameter bits can drastically alter the model behaviors, resulting in a sharp drop in accuracy or misprediction in the attacker's desired manner. (2) *Code-level attacks* [26], [6]: the attacker targets the functional code of the deep learning framework, including the executables generated by the compilers, or the underlying computation libraries. By flipping critical bits in these foundational components, the attacker can hijack the control flow of the inference execution, leading to large performance degradation or even system failure.

In spite of existing efforts to mitigate BFAs, it is still challenging to design a holistic solution to provide comprehensive protection over various threat scenarios. As BFAs are commonly realized via exploiting certain hardware vulnerabilities, a straightforward strategy is to fundamentally mitigate those threats [16], [11], [8], [21], [33], [9], [19]. While effective, these solutions often require specialized hardware upgrades, which can introduce compatibility issues in existing systems and limit their broader adoption. Besides, they lack generalization across different computing platforms and scenarios.

Due to the above limitations, researchers have been actively seeking for new solutions dedicated to the protection of DNN models against BFAs. Existing approaches can be classified into two categories. (1) *Redesigning the DNN model for better robustness*. This is normally achieved via model architecture modifications or retraining. Typical examples include the dynamic multi-exit architecture [45], weight reconstruction [25], quantization [13], [40], obfuscations of channels [34], [10] or bits [29]. (2) *Runtime integrity checking*. This strategy continuously monitors the DNN inference, detects and corrects potential errors caused by flipped bits. Some works introduce checksum [24], [7], hash functions [17], [18], or checker DNNs [28] for integrity verification. Other studies focus on the detection of critical bits as BFA targets [14], [31], [30].

Unfortunately, these approaches still exhibit several drawbacks in practice. (1) **High access requirement to the DNN models**. Some solutions require significant changes over the model parameters or network designs [45], [25], [13], [40], [34], [10], [29], which are often difficult for predeployed models or legacy systems. Some solutions require the access to the original training data [7], which may not be always feasible. (2) **Non-negligible runtime overhead**. Runtime integrity checking incurs additional latency for the inference execution, making it impractical for applications with strong real-time requirements [28], [31]. For example, on the CIFAR-10 dataset, DeepDyve [28] incurs a 9.1% computational overhead, while NeuroPots [31] adds a 9.7% time overhead. (3) **Degradation of model functionality**. So-

lutions that modify model parameters often compromise model performance while providing protection [31], [3], [45], [13], [40]. As reported in prior work [7], [45], BIN [40] reduces model accuracy by approximately 4.1%, RA-BNN [40] by 2.9%, NeuroPots [31] by 1.4%, and Aegis [45] by 1.2%. (4) **Lack of generalization**. More importantly, these solutions only target one specific attack exclusively. The majority of the works focus on the bit flips in model parameters [45], [25], [13], [40], [34], [10], [29], [24], [17], [18], [14], [31], [30]. To our best knowledge, there is only one work targeting the DNN executable-level threat [7], and no existing defense against bit flips in the computation libraries. However, its effectiveness is inconsistent across datasets and models—for example, on CIFAR-10, the attack still achieves a 24.8% success rate. Furthermore, since this approach relies on detecting anomalies in gradients, it is prone to false alarms when facing out-of-distribution inputs or adversarial examples.

Driven by the above limitations, we propose `ObfusBFA`, a novel and holistic methodology to mitigate different levels of BFAs efficiently. Its key insight is based on our observation that all BFAs rely on precise identification of vulnerable bits, which constitute a very small percentage (less than 0.01%) of the total bits (Section III-B). In contrast, flipping random bits can rarely impact the model inference [10]. Inspired by this, `ObfusBFA` aims to **degrade any carefully-orchestrated BFA into random flips, rendering the attack ineffective**. This is achieved by introducing obfuscation (i.e., random dummy operations) into the application architecture at different levels, including the model parameters, executables, and deep learning libraries. Such obfuscation has minimal impact on the inference results or speed, but alters the memory address offsets of vulnerable bits, thereby invalidating the attacker's ability to target specific bits. Note that model-level obfuscation has been applied to mitigate side-channel attacks [23], [51]. We repurpose this concept and extend it to multiple system layers for integrity protection against BFAs.

`ObfusBFA` is designed as an end-to-end protection framework to automatically provide comprehensive protection over a given DNN application (Figure 1). And `ObfusBFA` offers several key advantages compared to existing methods. First, it provides protection against both model-level and code-level BFAs on either quantized or floating-point models. Second, injecting dummy operations has zero impact on the model prediction accuracy, and minimally affects the inference latency and resource usage, making `ObfusBFA` a utility-preserving and lightweight solution. Third, `ObfusBFA` does not require model retraining or access to training data, rendering it more practical for real-world deployment.

II. Background

A. Bit-Flip Attacks against DNN Models

Bit-Flip Attacks (BFAs) are hardware fault injection attacks that corrupt memory by flipping bits, often via Rowhammer [20]. In deep learning, they target critical data to compromise predictions, typically in two forms: (1) *Untargeted BFAs*, which degrade overall accuracy, and (2) *Targeted BFAs*, which manipulate outputs for specific inputs (e.g., a chosen class or trigger) while leaving others unaffected. Both rely on bit-search algorithms to identify vulnerable locations, which are then flipped. BFAs operate at two levels:

- **Model-level Attacks.** The most common strategy flips bits in *model parameters*[4], [36], [37], [48], [38], [1]. TBT[37] targets critical neurons in the final layer and embeds a backdoor via bit flips. ProFlip [4] uses JSMA to identify vulnerable bits across layers. TA-LBF [1] formulates the search as a binary integer programming problem solved with ADMM.
- **Code-level Attacks.** A newer class flips bits in *executables or computation libraries*. Compilers (e.g., TVM [5], Glow [41]) translate models into binaries, which often rely on libraries like BLAS for GEMM operations. Flipping vulnerable bits in executables or libraries can hijack control flow. Chen et al.[6] showed flips in the `.text` section of executables can sharply reduce accuracy. Li et al.[26] proposed *FrameFlip*, where a single flipped bit in a jump opcode within OpenBLAS corrupts all dependent DNN applications.

III. Overview

A. Threat Model

In line with prior works [31], [48], [36], [26], we adopt the conventional threat model of BFAs. The attacker is an unprivileged user who shares the same physical machine as the victim's DNN application. Note that this co-location condition is a requirement for launching BFAs, not for our defense. Through techniques like Rowhammer, the attacker can flip the victim's data and code in the DRAM.

As a defense-focused work, we strategically consider a strong attacker, who has common knowledge about the victim's DNN application. (1) As the victim may employ some popular open-source DNN models, the attacker knows the DNN architecture, model weights, gradients and training data. This allows him to precisely target the flip bits within the original DNN model. (2) The victim may also use common deep learning frameworks to support the model inference. Therefore, the attacker knows the framework, dynamic libraries linked to the model, deep learning compilers and compilation settings. (3) The victim employs our `ObfusBFA` to transform the original DNN model and code executables. The attacker knows the detailed mechanism of `ObfusBFA`, but not the random numbers generated on-the-fly by the victim. The attacker also does not have access to the transformed DNN model or code executables.

Defense Scope. Consistent with the scope of most defense-oriented studies, we assume the attacker is non-privileged. An adversary with full system privileges—capable of directly reading or writing process memory—would not need to rely on hardware-induced faults such as RowHammer, since they could trivially modify memory contents. In such cases, BFAs are meaningless, and corresponding defenses unnecessary.

We focus on BFAs that rely on **precise bit-level manipulations**. These attacks fundamentally depend on the attacker's ability to identify and flip critical bits at certain physical

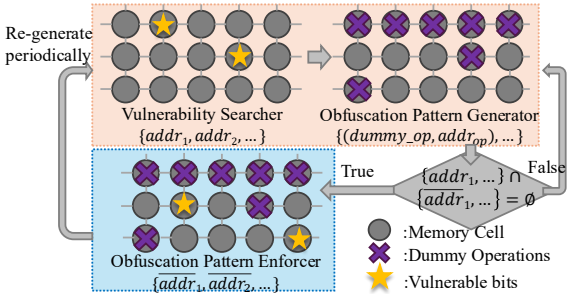

Fig. 1. Overview of ObfusBFA.

memory locations. Attacks that flip arbitrary random bits in weights or biases without modeling the underlying hardware feasibility fall outside the classical definition of BFAs and thus beyond the intended defense scope of existing BFA studies.

B. Design Insight

The design of our ObfusBFA is grounded in the following observation: **random bit-flips have minimal impact on the model inference process**, which is universal for different levels of the DNN system. Specifically, Cheng et al. [10] showed that randomly flipping bits in DNN's model weights have negligible effects on its prediction results, even when more bits are changed. We extend this conclusion to the code level by flipping each of all conditional jumps to its opposite control flow in a compiled OpenBLAS library. Out of 154,554 conditional jumps under the `-O2` optimization flag, only 85 jumps cause a drop in model accuracy when running ResNet-50 on the ImageNet dataset, while another 148 jumps lead to crashes or timeouts. The remaining 99.8% of jumps, despite being flipped, run without errors or performance drops. These results suggest that both DNN model weights and codebases are naturally resilient to random bit-flips in most cases.

This inspires the design of ObfusBFA: by introducing obfuscation into the model's inference (i.e., randomizing the memory address offsets for vulnerable bits), we can transform an organized BFA into random bit-flips, which can effectively alleviate the attacker's impact on the model results. Following this unified principle, ObfusBFA delivers protection over DNN applications against BFAs at both model and code levels.

ObfusBFA is designed to be attack-agnostic: it does not depend on attack-specific heuristics. Instead, it targets the attacker's fundamental assumption: the stable mapping between model elements and physical memory addresses, and breaks it by inserting randomized obfuscation primitives and periodically re-templating the layout. Any bit-searching scheme (gradient-based, reinforcement learning, heuristic search, etc.) that relies on consistent address/offset mappings will face the same barrier: critical-bit addresses are randomized and therefore an attacker's discovered offsets become unreliable.

IV. Methodology

A. Overall Workflow

Figure 1 shows the pipeline of ObfusBFA. It consists of three logical components: *Vulnerability Searcher*, *Obfuscation Pattern Generator*, and *Obfuscation Pattern Enforcer*.

- **Vulnerability Searcher**. This module scans the target application, and records the locations of all identified vulnerable bits. For different levels of BFAs, we design the corresponding algorithms to scan the target operation set (i.e., model weights for model-level BFAs, binaries for code-level BFAs) and identify bits that can significantly impact the model inference when flipped. Vulnerable bits that only cause a crash are not considered critical. The locations (offsets) of these vulnerable bits are recorded in a list.
- **Obfuscation Pattern Generator**. Based on these vulnerable bit locations, this module aims to produce the randomized memory address offsets of the target application. The detailed process is outlined in Algorithm 1. Specifically, after *Vulnerability Searcher* obtains the addresses of all vulnerable bits (Line 3), *Obfuscation Pattern Generator* processes the target system and inserts a random number of dummy operations before the critical elements (e.g., operations in the code, layers in the model architecture).

Algorithm 1 Obfuscation Pattern Generation

```
1:  Input: ElemSet, Prob          ▷ Target set, Obfuscation Probability
2:  Output: ObsPat                ▷ Obfuscation pattern
3:  ObsSuccess ← 0, VLoc ← getVulAddr(ElemSet)
4:  while ObsSuccess = 0 do
5:      for Elem ∈ ElemSet do
6:          if getRand ≤ Prob  or  Elem ∈ VLoc then
7:              (DummyOp, InsLoc) ← InsDummy(Op)
8:              ObsPat.append(DummyOp, InsLoc)
9:      VLocNew ← getVulAddr(ObsPat, ElemSet)
10:     if NoCommonAddr between VLoc and VLocNew then
11:         ObsSuccess ← 1
```

There are two points that need to be emphasized. First, an adaptive attacker may try to flip the bits not in the vulnerable list, attempting to bypass our defense. Such attack will be much less effective as those bits have minor impact on the model behaviors. Nevertheless, we also offer protection for all the other bits not in the list. Note that we cannot enforce the same degree of obfuscation as for the critical bits, which can terribly affect the model performance. Instead, we insert dummy operations following a preset probability, which is typically lower to reduce the overhead. This random insertion mechanism can help further reduce the effectiveness of potential attacks targeting the bits not classified as vulnerable, making them impractical.

Second, for the generated obfuscation pattern, we run *Vulnerable Searcher* again to check new vulnerable bits (Line 9). If there is no overlap between the new and original lists of vulnerable bit addresses, the obfuscation is successful. This ensures no vulnerable bits have relocated to new addresses that are also vulnerable. Otherwise, we run *Obfuscation Pattern Generator* again until the condition is met.

- **Obfuscation Pattern Enforcer.** At runtime, this module applies the generated obfuscation pattern to the model inference. Due to the insertion of dummy operations, the obfuscated operation set has a different memory layout from the unprotected one. However, this will not affect the model functionality,

as the inserted dummy operations have no impact on the target operation set. With the randomized memory addresses for critical operations, the attacker cannot identify the actual memory offsets of the vulnerable bits, effectively transforming any sophisticated BFA into an ineffective random BFA.

End-to-end Protection. Given a DNN system, we execute the above three components to generate the obfuscated versions of the model architecture, executable and libraries, and then launch them for robust inference. Note that a sophisticated attacker may try to reverse engineer the obfuscated code, and then adaptively identify new vulnerable bits for flipping. To mitigate this threat, we can periodically run the entire pipeline to generate and launch new obfuscated executables and libraries, e.g., every 10 minutes. Considering the long time for the attacker to flip bits (e.g., multiple hours with Rowhammer), this update frequency is sufficiently secure to mitigate the adaptive BFAs. As our defense pipeline is computationally lightweight and typically completes within a few seconds, the update has negligible impact on the overall performance. We can further launch multiple inference instances, and alternatively update them to reduce the impact on service availability.

Below we present the detailed mechanisms of each component for different levels of BFAs.

B. Vulnerability Searcher

We design new algorithms to search vulnerable bits in different levels of BFAs, respectively.

1) Model-level BFAs.: We employ a gradient-based ranking algorithm to identify a set of vulnerable bits in the model weights. For the weights in the *i*-th layer, given the loss function $\mathcal{L}$ and the weight matrix $\mathbf{W}$, we select the top-*k* weights from both the convolutional layers and the linear layers based on the magnitude of their gradients for the testing dataset. For convolution layers and linear layers from 1 to *z* in the model, the process of selecting the top-*k* most vulnerable weights can be expressed as:

$$\text{Top}_k\left[||\frac{\partial \mathcal{L}}{\partial \mathbf{W}}||\right] = \text{Top}_k\left[||g_1^{1,1}, g_i^{1,1}, \ldots, g_z^{M,N}||\right] \quad (1)$$

The locations of these top-*k* weights form the list of vulnerable locations. Since DNN models are loaded into memory in a sequential page-by-page manner, where each page has a fixed size and stores the weights contiguously, we can map the vulnerable weights directly to their index in memory from the beginning of the model. This indexing allows us to efficiently reference and protect specific weights during our defense implementation. It is important to note that this module is also applicable to real-world quantized models. During backpropagation, each layer can be temporarily dequantized to compute the gradients and then re-quantized afterward. This solution has been widely adopted in prior BFA studies [4], [36], [37], [48], [38], [1].

2) Code-level BFAs.: We aim to search for the critical jump instructions in executables and shared libraries. We focus on jump opcodes as they are highly susceptible to BFAs and flipping a single bit can invert their semantic condition (e.g., changing jump-if-equal to jump-if-not-equal), effectively reversing the control flow. Hence, they are the main target of existing code-based attack [26]. The existence of other vulnerable instructions and their sensitivity to bit flips is not explored. To address this uncertainty, `ObfusBFA` also obfuscates the addresses of non-critical elements not in the vulnerable list, as described in Section IV-A.

Specifically, we iterate through all opcodes in the binary files. If the current opcode is a conditional jump, we flip the conditional jump to its semantic opposite and generate a new library file. In the x86 instruction set, each conditional jump has a short jump and near jump version, which have different ranges and opcodes. For example, `je` has the short jump opcode `0x74` and the near jump opcode `0x0F84`. The algorithm considers both cases. We then use the instrumented binary to run inference and test accuracy. If the accuracy drops, the branch is considered vulnerable, and the corresponding bit address is recorded.

C. Obfuscation Pattern Generator

We develop new obfuscation solutions to randomize the memory offsets of the identified critical operations.

1) Model-level BFAs.: We introduce two strategies to obfuscate the model address layouts.

- **Inserting Dummy Layers.** This strategy involves inserting an additional computational layer immediately after the activation function φ of a given layer. The insertion of a dummy layer L_i should not affect the output of the original layer.

For linear layers, the dummy layer L_i is an identity matrix of the same dimensions as the input to the layer. For convolutional layers, where the input and output have *c* channels and kernel size of (k_1, k_2), the dummy layer is initialized as:

$$L_i^{(a,b,d,m)} = \begin{cases} 1 & \text{if } a = b = 1 \text{ and } d = m \\ 0 & \text{otherwise} \end{cases}$$

where a, b are the indices of the row and column, d is the index of the input channel, and m is the index of the filter. We choose a kernel size of $k_1 = k_2 = 1$, which minimizes the extra computation introduced by the dummy layer.

- **Inserting Dummy Neurons.** Another obfuscation strategy is the insertion of dummy neurons into existing layers. These dummy neurons ensure that the prediction behavior of the original DNN remains unchanged after their addition.

To insert *n* dummy neurons into a current computation layer with weights $\mathbf{W_i^{(b,a)}}$, assuming the next computation layer has weights $\mathbf{W_{i+1}^{(c,b)}}$, then given the input $\mathbf{X_i}$, the forward process is formulated as:

$$\mathbf{W_{i+1}^{c,b}} \cdot (\mathbf{W_i^{b,a}} \cdot \mathbf{x}) = \mathbf{W_{i+1}^{c,(b+n)}} \cdot (\mathbf{W_i^{(b+n),a}} \cdot \mathbf{x}) \quad (2)$$

To ensure the injected neurons are non-functional (i.e., "dummy"), their weights are set to vectors of all zeros with biases of zero. Since adding dummy neurons alters the dimensions of the current layer, the subsequent layer must also have the corresponding dummy neurons added to maintain

the dimensional consistency. Specifically, for a convolution layer, we add n extra filters to the current layer, increasing the number of output channels by n. As a result, the number of input channels of the next convolution layer must also be increased by n, with the additional channels initialized to zeros. Similarly, for each linear layer, we increase the number of output features by n, and adjust the number of input features of the next linear layer by n, with the added parameters initialized to zeros.

`ObfusBFA` can be extended to defend modern large language models (LLMs) against BFAs. LLMs differ significantly from CNNs in both computational graph and memory organization, posing new challenges for maintaining obfuscation effectiveness while controlling overhead. To accommodate these differences, we adapt `ObfusBFA` to selectively inject obfuscation operations into the feed-forward MLP modules of transformer-based architectures. Specifically, dummy neurons are inserted within the linear layers of the MLP blocks but excluded from attention modules to prevent interference with the key–value (KV) cache reuse and sharing mechanisms critical to efficient autoregressive decoding. This selective design ensures that the obfuscation perturbs the model's internal dataflow sufficiently to disrupt attacker bit-localization while preserving overall throughput and memory access efficiency. For earlier LLM architectures with sequential MLP layers, `ObfusBFA` can be directly applied without modification. In contrast, for modern architectures such as Qwen and LLaMA that feature parallel MLP branches (e.g., gated or SwiGLU variants), the dummy neurons are inserted symmetrically into both parallel linear branches as well as the subsequent merging linear layer following element-wise multiplication. This ensures that the obfuscation pattern remains consistent and uniformly distributed across computation paths, preventing the attacker from isolating specific critical bits in one branch.

2) Code-level BFAs.: To add obfuscation at the code level, our strategy is to insert dummy operations (e.g., *DummyOP*) before the critical instructions. The `NOP` (No Operation) instruction in `x86` architecture is a one-byte instruction that takes up space in the instruction stream but performs no useful operation. It does not modify any machine state except for advancing the `EIP` (Extended Instruction Pointer) register. As a result, after we determine that a specific location requires obfuscation, a random number of `NOP` instructions are inserted before the current opcode, thereby altering the memory layout without changing the functionality.

To implement this memory address randomization efficiently, we develop an LLVM-based backend obfuscation pass. While it is theoretically possible to insert `NOP` at any compilation stage, it is challenging to do so in earlier stages (e.g., at the intermediate representation (IR) level or in source code). Since the attack targets the memory image, corresponding to the final binary, it is important to ensure precise control over the location of inserted `NOP` instructions in the binary. However, locating the exact binary position for vulnerability obfuscation from the IR level is difficult due to the abstraction gap between the IR and the final binary. Inserting `NOP` instructions at the source code level also presents challenges. Even if the mapping between the source code and binary is known, compiler optimizations can relocate or even remove the inserted code, making it difficult to control where the `NOP` instructions will eventually appear in the binary.

To overcome these challenges, we insert `NOP` instructions at the lower-level representation, specifically after the compiler performs all optimizations and just before the target code is emitted . This step corresponds to the `addPreEmitPass2` in LLVM, ensuring that the inserted `NOP` instructions appear precisely where required in the final binary.

D. Obfuscation Pattern Enforcer

1) Model-level BFAs: At runtime, we load the target model and apply the obfuscation patterns to disrupt the memory layout. Since the obfuscation involves only simple operations, e.g., padding zeros to the weights or inserting non-functional neurons, it is computationally efficient and incurs minimal overhead. As a result, the transformation is applied almost instantaneously, leaving very little time for an attacker to exploit the model via Rowhammer or other bit-flip techniques before the obfuscation is completed.

Moreover, due to the design of the obfuscation strategies, the obfuscated model maintains identical functionality to the original model. This ensures that the obfuscation process does not introduce any unintended changes to the model's behavior, preserving its accuracy and performance while simultaneously protecting it against memory-based attacks. Additionally, the memory layout of the model is randomized with each load, making it even more challenging for an attacker to predict the specific bit locations they need to target. This randomness further strengthens the defense.

2) Code-level BFAs: Once the instrumented library or executable with the altered memory layout is generated, we load them for runtime model inference. In Linux environments, when a program is loaded and executed, the dynamic linker (ld-linux.so) first searches the libraries listed in the `LD_PRELOAD` environment variable for any undefined references before searching other libraries. Therefore, we set `LD_PRELOAD` to our obfuscated versions, which can intercept and redirect function calls to utilize the new libraries. For code executables, we modify the compilation process, e.g., instruct TVM to generate LLVM bitcode, enabling our custom LLVM backend pass to apply the obfuscation techniques.

We can frequently regenerate libraries and executables to enhance protection. The compilation cost can be very small with the following optimization. Since the functions prone to BFAs are concentrated in a small subset of source files (8 in OpenBLAS), we only need to recompile these files instead of the entire project. To further reduce the compilation time, we pre-generate LLVM intermediate representation (IR) and perform binary generation from this IR, as the obfuscation is applied in the backend, specifically in the IR-to-binary translation process. Our tests show that the compilation process takes around 0.683 seconds, and generating the dynamic library adds

just 0.103 seconds. This enables frequent, low-cost binary updates, maximizing the runtime protection against BFAs.

V. Security Analysis

A. Discussion of Potential Threats

We provide analysis about the security robustness of ObfusBFA from different perspectives.

• **De-obfuscating** ObfusBFA. We acknowledge that an attacker who can repeatedly observe randomized executions or gather high-quality side-channel traces may attempt statistical denoising to reduce randomness. However, the denoised result would at best reveal the pre-obfuscation layout, which is already known by the attacker in our threat model. Because the obfuscation patterns in ObfusBFA are periodically refreshed with unpredictable randomness, any previously gathered information becomes obsolete, significantly increasing the cost and difficulty of long-term exploitation.

• **Targeting a different set of bits**. Theoretically, *Vulnerability Searcher* can identify majority of vulnerable bits, and attacker's identified bits are highly likely to fall within this set. Even if the attacker selects a different set of bits to bypass the defense (with less attack effectiveness), our design (Algorithm 1) ensures obfuscation is applied probabilistically throughout the entire codebase or model weights. It will largely alter the memory offsets to thwart the attack.

• **Recovery of critical bits.** While inserting dummy operations or NOPs may slightly modify the side-channel signature during execution, it does not offer a practical pathway for attackers to recover exact locations of critical weights or bits for the following reasons. (1) Existing side-channel attacks on ML accelerators operate at coarse architectural granularity (e.g., layer or kernel level) and lack the precision to infer bit-level information required for BFAs [49]. (2) In the threat model of BFAs, the attack is assumed to already know precise bit locations in the victim model before obfuscation. Since ObfusBFA inserts only a small fraction of dummy operations without altering the original memory layout, no additional information about sensitive bit positions is revealed. Even if attackers could approximate hotspot regions through side-channel observations, mapping them to exact bits remains infeasible due to the physical constraints of RowHammer and memory addressing. (3) ObfusBFA periodically randomizes obfuscation patterns, invalidating any previously collected traces. Our reproduced model extraction attack [47] further confirms that side-channel traces contain negligible information about bit locations (achieving only 0.7% recovery accuracy), validating ObfusBFA does not increase leakage.

B. Limitation

We mainly perform empirical analysis and experimental evaluations about the security of ObfusBFA. Deriving a theoretical bound on all possible risks is an interesting but highly challenging problem, as it requires a complete characterization of all possible hardware fault mechanisms, binary layouts, and memory-access behaviors, many of which are inherently platform-dependent and stochastic. Such formalization is therefore beyond the scope of current BFA literature, where existing state-of-the-art defenses (e.g., BitShield [7], LIMA [14], NeuroPots [31]) rely on empirical robustness evaluation rather than provable guarantees. While a closed-form theoretical bound is impractical given hardware nondeterminism, our extensive adaptive evaluations serve as a strong empirical upper bound on potential risks in realistic scenarios.

VI. Evaluation

A. Experimental Setup

Attacks under Evaluation. For *code-level BFAs*, we assess the effectiveness of ObfusBFA against two state-of-the-art, untargeted attacks, which are the only known attacks of their kind to date. (1) *FrameFlip* [26] targets the BLAS library by flipping conditional jumps in LLVM bitcode via an XOR operation at the IR level. We used PyTorch version 2.3.0 with OpenBLAS version 0.3.20. To better reflect real-world threats, we tested our defense on a C++-based inference infrastructure, as used in [26], to demonstrate its applicability beyond Python-based frameworks. (2) The attack in [6] targets the TVM compiler by manipulating vulnerable bits in DL executables to degrade overall model accuracy. We adopt the TVM version 0.18.dev0, and choose the -O3 optimization flag. We also include adaptive versions of both attacks to evaluate ObfusBFA's robustness under adversarial awareness.

For *model-level BFAs*, we evaluate ObfusBFA against one untargeted attack [36], and three targeted attacks for CV models: T-BFA [38], TBT [37], TA-LBF [1], and OneFlip [27]. We also evaluate one BFA targeting large language models (LLM) [46]. As with the code-level setting, we also consider adaptive variants of these attacks.

Datasets and Models. Following common selections in existing attacks [26], [6], [4] and defenses [34], [31], [45], we evaluate ObfusBFA on three datasets: CIFAR-10 [22], German Traffic Sign Recognition Benchmark (GTSRB) [15], and ImageNet ILSVRC-2012 subset [42]. Since experiments conducted on CPUs can be time-consuming, we use a subset of the testing dataset for evaluation.

We choose popular network architectures that are widely used for image classification tasks, including VGG-16 [44], ResNet-20, ResNet-32, ResNet-34, and ResNet50 [12].

Baselines. For *code-level BFAs*, we compare our ObfusBFA with the only known defense BitShield [7]. For *model-level BFAs*, we compare ObfusBFA against several state-of-the-art defenses targeting both untargeted and targeted bit-flip attacks on model weights. Specifically, for untargeted BFAs, we choose DeepShuffle [34], NeuroPots [31], HammerDodger [10], and RREC [29]. For targeted BFAs, we choose BIN [13], RA-BNN [40], and Aegis [45].

B. Results for Code-level BFAs

Model Utility Evaluation. A key metric for evaluating any defense is its ability to preserve the model's utility, particularly in terms of accuracy (Acc). Tables I and II report the accuracy results after applying ObfusBFA across different DL frameworks (C++-based and PyTorch-based) with the TVM compiler.

TABLE I
MODEL UTILITY EVALUATION (DL INFRA.)

DL Framework	Dataset-Model	Base Acc	ΔAcc (%)
C++ (O1–O3)	CIFAR10-ResNet20	97.00	0
	GTSRB-VGG16	95.31	0
	ImageNet-ResNet50	91.67	0
PyTorch (O1–O3)	CIFAR10-ResNet20	81.25	0
	GTSRB-VGG16	93.81	0
	ImageNet-ResNet50	87.50	0

Since we observe no accuracy difference across optimization levels (O1–O3), we merge the results for simplicity in Table I. The "Base Acc" column shows the model's accuracy without `ObfusBFA`, while the "ΔAcc" column represents the change in accuracy after applying `ObfusBFA`. The "Flag" column indicates the optimization flag used during library compilation. The results demonstrate that `ObfusBFA` does not degrade model performance, with no observable accuracy loss across all datasets and models tested.

TABLE II
MODEL UTILITY EVALUATION (DL COMPILER TVM)

Dataset-Model	Base Acc (%)	ΔAcc (%)
CIFAR10-ResNet20	80.50	0
GTSRB-VGG16	98.50	0
ImageNet-ResNet50	85.00	0

Mitigating Untargeted Attacks. For untargeted BFAs, we measure the average accuracy drop across the identified vulnerable bit set. Tables III, IV, and V report the number of identified vulnerable bits (#Vuln. Bits), the original model accuracy (before the attack), and the accuracy both without and with the defense applied to the vulnerable bit set. These tables demonstrate that `ObfusBFA` successfully restores the model to its original accuracy for both C++-based and PyTorch-based DL infrastructures under the *FrameFlip* attack, and for the TVM compiler under the attack in [6].

TABLE III
MITIGATING UNTARGETED ATTACKS (C++ INFRA.)

Flag	Model	#Vuln Bits	Acc (%) Base	w/o. Def	w. Def
O1	ResNet20	80	97.00	33.90	97.00
	VGG16	61	95.31	18.00	95.31
	ResNet50	81	91.67	29.10	91.67
O2	ResNet20	85	97.00	36.50	97.00
	VGG16	64	95.31	22.50	95.31
	ResNet50	85	91.67	31.20	91.67
O3	ResNet20	85	97.00	36.20	97.00
	VGG16	63	95.31	23.50	95.31
	ResNet50	85	91.67	31.60	91.67

TABLE IV
MITIGATING UNTARGETED ATTACKS (PYTORCH INFRA.)

Flag	Model	#Vuln Bits	Acc (%) Base	w/o. Def	w. Def
O1	ResNet20	34	81.25	23.67	81.25
	VGG16	30	95.31	6.88	95.31
	ResNet50	28	87.50	5.58	87.50
O2	ResNet20	34	81.25	25.14	81.25
	VGG16	34	95.31	11.81	95.31
	ResNet50	29	87.50	6.25	87.50
O3	ResNet20	34	81.25	25.18	81.25
	VGG16	31	95.31	8.06	95.31
	ResNet50	28	87.50	7.14	87.50

Adaptive Attacks. We consider an adaptive attacker aware of our defense mechanism but not the on-the-fly obfuscated parameters. Since `ObfusBFA` introduces randomness into the memory layout obfuscation, the attacker's only viable option to circumvent it is to flip more bits. To this end, he flips not only the bits in the identified vulnerable bit list but also adjacent bits around each vulnerable bit address *addr*. Given that code alignment is typically enforced by compilers (e.g., aligning instructions to 16-byte boundaries for efficient CPU instruction fetching), `ObfusBFA` may cause 16*n*-byte latencies in subsequent function. Therefore, the attacker can flip all bits in the following range for each vulnerable bit address *addr*:

$$[addr - x_1 + x_2 \cdot al, addr + x_1 + x_2 \cdot al] \tag{3}$$

where *al* represents the alignment size set during compilation, and x_1 and x_2 are parameters that control the flipping range. The larger value results in a broader range of bit flips.

We conducted such adaptive attack on all three dataset-model configurations using the compile flags `-O1`, `-O2`, and `-O3` on the C++ deep learning framework and the DL compiler TVM. For each vulnerable location *addr*, we set the alignment size *al* to 16, and evaluate x_2 in the range [0, 1, 2] for each evaluation, while x_1 takes values from the set {5, 10, 15, 20, 25, 30, 35, 40, 45} at each time. We observed an increase in the percentage of program timeouts or crashes, but no accuracy drop was detected across any experimental setting. These results demonstrate that even when an attacker is aware of our defense mechanism, he is unable to bypass it. The adoption of randomized memory address obfuscation turns carefully targeted bit flips into random flips, neutralizing the attacker's adaptive strategy and preserving model accuracy.

TABLE V
MITIGATING UNTARGETED ATTACKS (TVM COMPILER)

Model	#Vuln. Bits	Acc (%) Base	w/o. Def.	w. Def.
ResNet20	94	80.50	17.98	80.50
VGG16	112	98.50	16.83	98.50
ResNet50	72	85.00	21.25	85.00

Overhead. A key objective of `ObfusBFA` is to minimize overhead. We evaluate time, storage, and memory costs, averaged over 10 inference runs. Tables VI shows the results. Across models and frameworks, `ObfusBFA` incurs negligible overhead: time overhead is mostly small (some even benefit from micro-optimizations), storage growth is marginal ($\leq$0.37%), and memory fluctuations remain minor (up to 1.69%). Overall, `ObfusBFA` preserves performance with minimal resource costs, ensuring practicality for deployment.

We also consider the latency as the number of protected bits grows. While protecting a larger number of critical weights may require additional attempts to determine valid dummy placements, the overhead of Obfuscation Pattern Generator remains negligible. The generator uses purely random sampling with lightweight validity checks, avoiding any iterative optimization process. In our implementation, generating a valid obfuscation pattern typically takes only within a seconds, and this cost occurs once per deployment rather than per inference. As shown in Figure 2, the generation time remains stable as the number of protected bits increases, indicating

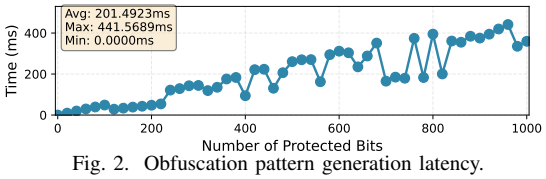

Fig. 2. Obfuscation pattern generation latency.

minimal scalability concerns. Furthermore, the generation process can be parallelized or performed offline, making the overall overhead negligible in practical deployments.

TABLE VI
TIME AND STORAGE OVERHEAD FOR C++ INFRA.

Flag	Model	Insert Count	ΔTime (ms/%)	ΔStorage (KB/%)	ΔMemory (KB/%)
O1	ResNet20	63636	+2(3.04%)	+47(0.37%)	-13(-0.46%)
	VGG16	34664	-1(-1.08%)	+27(0.22%)	+4(1.00%)
	ResNet50	37085	+14(1.31%)	+27(0.22%)	+195(1.00%)
O2	ResNet20	55667	-8(-13.2%)	+39(0.29%)	+30(1.05%)
	VGG16	4939	-4(-3.13%)	+4(0.03%)	-7(-1.67%)
	ResNet50	33494	-42(-3.90%)	+23(0.17%)	-1,132(-6.01%)
O3	ResNet20	56393	-8(-12.8%)	+39(0.26%)	+67(2.36%)
	VGG16	5867	-2(-1.26%)	+4(0.03%)	+7(1.69%)
	ResNet50	5272	+51(5.02%)	+4(0.03%)	-153(-0.80%)

Comparison. We compare `ObfusBFA` with BitShield [7], the only existing defense at the DNN executable level. We use the ResNet50 model on ImageNet for the BFA against the TVM compiler. Both defenses maintain the model's original accuracy (ΔAcc = 0%). However, BitShield introduces a notable 8.22% runtime overhead, while `ObfusBFA` incurs a slight performance improvement of -0.55%, possibly due to compiler-level code alignment optimizations introduced by obfuscation. In terms of mitigation effectiveness, `ObfusBFA` completely thwarts the attack with a 100% mitigation rate, while BitShield achieves 97.9%. These highlight that `ObfusBFA` not only offers superior protection but also improves efficiency in certain cases.

Fine-grained Overhead and Cache Analysis. To further analyze the performance impact of `ObfusBFA` at a finer granularity, we quantify per-neuron overhead and cache behavior under varying protection densities.

First, we measure the per-neuron and per-parameter overhead by normalizing the number of inserted obfuscation elements per layer to the total parameter count and computing their corresponding latency contribution during inference. As summarized in Table VII, the insertion density remains below 0.5% even for the smallest models, indicating that the additional operations constitute only a negligible fraction of total parameters. Empirically, this translates to less than 1% latency overhead per inference on average, confirming that the lightweight obfuscation density used in our experiments imposes minimal per-neuron cost.

TABLE VII
PER-NEURON INSERTION DENSITY AND ESTIMATED OVERHEAD.

Dataset-Model	Parameter amount (bit)	Density (%)
CIFAR10-ResNet20	8,725,504	0.5
GTSRB-VGG16	477,536,096	0.2
ImageNet-ResNet50	817,840,576	0.1

Second, to investigate potential cache-level implications, we profile the cache behavior of `ObfusBFA` using `perf` on Linux. We monitor hardware events such as `cache-misses` under different protection densities. The results, illustrated in Figure 3, show that the current protection density achieves strong security coverage with minimal cache-side impact.

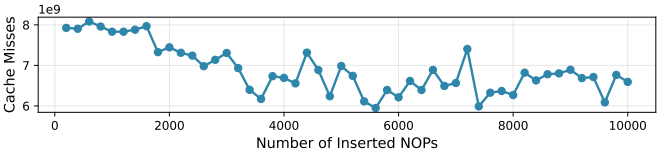

Fig. 3. Cache miss rate.

To mitigate this overhead, two practical optimizations can be adopted: (i) aligning dummy insertions with cache-line boundaries to reduce fragmentation; (ii) grouping insertions within local memory regions to preserve cache locality and minimize access overhead.

Generality across Different Code-level Attack Scenarios. To demonstrate the generality of `ObfusBFA` beyond the TVM and OpenBLAS setups, we extend our experiments to two additional configurations: (1) statically linked OpenBLAS integrated directly into the victim application, thus eliminating dynamic linking; (2) an alternative math library, FBGEMM, commonly used in deep learning inference workloads.

The results, summarized in Table VIII, show that `ObfusBFA` consistently reduce the success rate of functional bit-flips by more than 90% across both libraries, confirming its portability and robustness against diverse backend implementations. These findings validate that the obfuscation and runtime protection mechanisms offered by `ObfusBFA` are not tied to a specific software stack but can be effectively applied to different code-level execution environments.

TABLE VIII
CROSS-LIBRARY EVALUATION OF `ObfusBFA`.

Library	Model	#Vuln Bits	Acc (%) Base	w/o. Def	w. Def
OpenBLAS (Static)	ResNet20	77	97.00	31.66	97.00
	VGG16	61	95.31	18.91	95.31
FBGEMM	ResNet20	81	96.33	21.34	96.33
	VGG16	59	93.86	17.32	93.86

C. Results for Model-level BFAs

Model Utility Evaluation. We applied `ObfusBFA` to model-level BFAs and evaluated its impact on model functionality, specifically in terms of accuracy. The results, shown in the Table IX, indicate that `ObfusBFA` barely degrades the performance of the original model.

TABLE IX
MODEL UTILITY EVALUATION AFTER APPLYING `ObfusBFA`.

Dataset-Model	Base Acc (%)	ΔAcc (%)
CIFAR10-ResNet20	85.36	+0.01
CIFAR10-ResNet32	84.38	+0.01
CIFAR10-ResNet34	85.70	0
ImageNet-ResNet50	75.84	0
GTSRB-VGG16	84.26	-0.02

Mitigating Untargeted Attacks. We assess the effectiveness of `ObfusBFA` in mitigating the untargeted BFA [36]. The results in Table X show a significant accuracy drop for models without `ObfusBFA`'s protection (the *w/o. Def.* column).

Notably, the ImageNet-ResNet50 model's accuracy is reduced to nearly zero. However, after applying ObfusBFA (the *w. Def.* column), all models demonstrate resilience, maintaining accuracy levels close to their original accuracy (*Base* in the table). These results confirm that ObfusBFA effectively mitigates untargeted attacks across a range of models.

TABLE X
MITIGATING UNTARGETED ATTACKS.

Model	#Flipped Bits	Acc (%)		
		Base	w/o. Def.	w. Def.
ResNet20	30	85.36	11.46	82.53
ResNet32	14	84.38	11.81	82.00
ResNet34	30	85.70	19.84	83.63
ResNet50	10	75.84	0.10	75.85
VGG16	30	84.26	18.14	84.39

Mitigating Targeted Attacks. Targeted attacks differ from untargeted BFAs in that they seek to induce specific misclassifications without significantly impacting the overall model performance. To assess the effectiveness of ObfusBFA in defending against targeted BFAs, we measure the attack success rate (ASR), which indicates the percentage of successful misclassifications made by the attacker.

Table XI presents the ASR before (w/o. Def.) and after applying ObfusBFA (w. Def.), along with the number of flipped bits during the attacks. ObfusBFA significantly reduces the ASR across all three types of targeted attacks.

We further validate the scalability and universality of ObfusBFA in protecting LLMs against BFAs. We evaluate ObfusBFA on three representative LLMs of varying scales: Qwen 1.5–1.8B, Qwen 1.5–4B, and LLaMA2–7B, against a state-of-the-art BFA targeting decoder-based LLMs, BitHydra [46]. The experimental results, summarized in Table XI, show that ObfusBFA successfully mitigates the attack's impact on all three LLMs, restoring the attack success rate to zeros. These findings confirm that ObfusBFA generalizes effectively across model families and remains robust even under new attack objectives specific to LLMs.

Adaptive Attacks. We consider a sophisticated attacker aware of ObfusBFA's obfuscation techniques (dummy layers/neurons) but unaware of exact locations. To bypass ObfusBFA, the attacker flips additional bits around each vulnerable address *addr* in the range $[addr - x, addr + x]$.

Tables XII and XIII show that ObfusBFA remains robust: for untargeted attacks, accuracy is largely preserved, and for targeted attacks, the attack success rate (ASR) stays very low. The defense's effectiveness arises from most parameters having negligible impact when flipped and the random placement of dummy operations, making brute-force flipping ineffective. Targeted attacks require precise bit selection, which is thwarted by ObfusBFA 's randomization.

Moreover, a stronger attacker attempting to distinguish dummy from real operations is impractical. ObfusBFA obfuscates both model and code levels, so observed side-channel variations cannot be attributed to a specific level (i.e, dummy layer, neuron, or operation). Any observed changes could equally stem from obfuscation in either source. Additionally, periodic refresh of random patterns renders any inferred layout quickly obsolete, making side-channel attacks ineffective.

TABLE XI
MITIGATING TARGETED ATTACKS.

Attack	Model	#Flipped Bits	ASR (%)	
			w/o. Def.	w. Def.
T-BFA	ResNet20	6	99.84	8.16
	ResNet32	3	99.67	21.4
	ResNet34	20	99.83	10.43
	ResNet50	7	99.99	0.11
	VGG16	20	100	1.81
TBT	ResNet20	97	93.99	2.22
	ResNet32	147	76.78	8.11
	ResNet34	130	77.49	6.04
	ResNet50	131	100	0
	VGG16	126	98.69	4.89
TA-LBF	ResNet20	9.19	100	1.5
	ResNet32	10.92	100	1.9
	ResNet34	19.45	100	1.5
	ResNet50	12.76	100	0
	VGG16	18.42	100	2.7
OneFlip	ResNet20	1	100	0
	ResNet32	1	100	0
	VGG16	1	100	0
BitHydra	Qwen1.5-1.8B	4	100	0
	Qwen1.5-4B	12	100	0
	Llama-2-7b	30	97.1	0

TABLE XII
EVALUATION FOR UNTARGETED ADAPTIVE ATTACK

Model	Acc (%)				
	x=3	x=5	x=7	x=9	x=11
ResNet20	76.11	80.20	82.14	82.14	82.14
ResNet32	75.05	76.62	76.24	75.97	75.64
ResNet34	85.29	82.41	81.73	83.32	83.13
ResNet50	74.36	75.84	75.83	75.83	75.87
VGG16	82.74	80.55	80.21	82.58	82.59

Overhead. The time and memory overhead incurred by ObfusBFA is shown in Figure 4, where "insert probability" denotes the likelihood of inserting dummy layers and neurons. We measure the percentage increase in both the time and memory required to run the CIFAR-10 test dataset using ResNet20, compared to an unobfuscated model. The reported values are averaged over 100 runs across all images in the

TABLE XIII
EVALUATION FOR TARGETED ADAPTIVE ATTACK

Attack	Model	ASR (%)				
		x=3	x=5	x=7	x=9	x=11
T-BFA	ResNet20	0	0	0	0	0
	ResNet32	1.83	1.83	1.83	1.83	1.83
	ResNet34	22.9	22.9	22.9	22.9	22.9
	ResNet50	0.11	0.11	0.11	0.11	0.11
	VGG16	1.55	1.55	1.55	1.55	1.55
TBT	ResNet20	2.21	2.22	2.21	2.21	2.21
	ResNet32	8.14	8.13	8.15	8.12	8.18
	ResNet34	6.04	6.04	6.04	6.04	6.04
	ResNet50	0	0	0	0	0
	VGG16	4.89	4.89	4.89	4.89	4.89
TA-LBF	ResNet20	1.5	0.99	1.98	1.98	1.98
	ResNet32	1.9	0.5	0.99	0.99	0.99
	ResNet34	1.5	1.49	2.97	1.98	1.98
	ResNet50	0	0	0	0	0
	VGG16	2.98	2.98	1.98	1.98	1.98

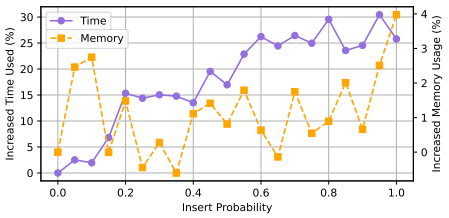

Fig. 4. Time and memory Overhead.

test dataset. While ObfusBFA incurs some time and memory overhead, it remains relatively low and acceptable. Notably, our experiments suggest that an insert probability below 0.3 is sufficient in most cases, effectively minimizing the overhead without compromising the defense effectiveness.

Comparison. First, we evaluate ObfusBFA against four untargeted defense schemes. The evaluation is performed using the ResNet20 model on CIFAR-10, and the results are presented in Table XIV. Since the source code for some of these schemes is unavailable, we rely on the data reported in their respective papers. The "-" symbol in the table indicates that a result was not reported in the corresponding paper. We observe that ObfusBFA vastly outperforms all other methods in terms of resilience against untargeted BFAs.

TABLE XIV
COMPARISON WITH UNTARGETED DEFENSE SCHEMES

Defense	ΔAcc (%)	ΔTime (%)	ΔRounds (%)	Mitigation Rate(%)
DeepShuffle	-	2.5	571	-
Neuropots	-1	9.7	-	93.3
HammerDodger	-	**0.8**	-	-
RREC	**+0.03**	5.8	1,452	-
ObfusBFA	+0.01	6.6	**160,831**	**100**

Next, we compare ObfusBFA with three targeted defense schemes. We adopt the ResNet32 model on CIFAR-10. The results are presented in Table XV, reporting the *ΔAcc (%)* (the change in accuracy after applying the defense) and *ASR (%)* (the percentage of successful attacks in both TBT and TA-LBF attack scenarios). We observe that ObfusBFA achieves the best performance, with a minimal accuracy drop (+0.01%) and significantly lower ASR in both TBT (8.11%) and TA-LBF (1.9%) scenarios. This indicates that ObfusBFA effectively reduces the success rate of targeted BFAs while better preserving accuracy than other defenses.

TABLE XV
COMPARISON WITH TARGETED DEFENSE SCHEMES

Defense	Δ Acc (%)	ASR (%)	
		TBT	TA-LBF
BIN	-2.26	94.8	100
RA-BNN	-1.71	74.5	100
Aegis	-1.26	19.9	6.3
ObfusBFA	**+0.01**	**8.11**	**1.9**

Resilience against Large-scale BFAs. To further evaluate the robustness of ObfusBFA under more aggressive attack scenarios, we include larger-scale BFAs. While the previous evaluation already covers configurations with over 100 flipped bits (e.g., Table V and Table XI), we now increase the attack magnitude to 150–160 bits using the untarget and target attack (T-BFA) attack. The new results are summarized in Tables XVI and XVII. Across various DNNs, including ResNet and VGG families, ObfusBFA maintains strong protection. This efficiency arises because the obfuscation operations inserted by ObfusBFA disrupt subsequent memory mappings and data offsets, breaking the linear relationship between the number of protected bits and latency overhead. Consequently, the protection cost does not grow proportionally with the number of flipped bits.

TABLE XVI
MITIGATING LARGE-SCALE UNTARGETED ATTACKS.

Model	#Flipped Bits	ΔTime (%)	Acc (%)		
			Base	w/o. Def.	w. Def.
ResNet20	113	5.8	85.36	10.54	80.32
ResNet50	156	7.2	75.84	0.10	69.45
VGG16	149	6.6	84.26	6.69	80.11

TABLE XVII
MITIGATING LARGE-SCALE TARGETED ATTACKS.

Model	#Flipped Bits	ΔTime (%)	ASR (%)	
			w/o. Def.	w. Def.
ResNet20	146	5.2	100	8.22
ResNet50	158	5.5	100	0.11
VGG16	162	7.3	100	1.90

It is worth noting that large-scale bit-flip attacks are often impractical on real hardware. It has been demonstrated that state-of-the-art RowHammer-based BFAs (e.g., DeepHammer [48]) are unlikely to succeed if the targeted bits in the model are simply numerous. Therefore, ObfusBFA remains sufficiently robust and efficient under realistic large-scale attack settings.

VII. CONCLUSION

We introduce ObfusBFA, an efficient and effective approach to mitigating the growing threat of BFAs targeting both model and code levels of DNN applications. By leveraging a novel obfuscation strategy that inserts randomized dummy operations, ObfusBFA effectively complicates the attacker's ability to locate and manipulate vulnerable bits, turning orchestrated attacks into random bit flips. Our evaluation shows ObfusBFA's robustness against both untargeted and targeted BFAs across a range of datasets and architectures. It also brings minimal computational and storage overhead, making it a practical solution for real-world applications.

VIII. ACKNOWLEDGEMENT

This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research & Development Programme (Development of Secured Components & Systems in Emerging Technologies through Hardware & Software Evaluation <NRF-NCR25-DeSNTU-0001>). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the view of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] 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. *arXiv preprint arXiv:2102.10496*, 2021.

[2] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. Practical fault attack on deep neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2204–2206, 2018.

[3] Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, and Farinaz Koushanfar. Deepattest: An end-to-end attestation framework for deep neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 487–498, 2019.

[4] Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Proflip: Targeted trojan attack with progressive bit flips. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7718–7727, 2021.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[6] Yanzuo Chen, Zhibo Liu, Yuanyuan Yuan, Sihang Hu, Tianxiang Li, and Shuai Wang. Unveiling signle-bit-flip attacks on dnn executables. *arXiv preprint arXiv:2309.06223*, 2023.

[7] 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.

[8] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[9] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 55–71. IEEE, 2019.

[10] Cheng Gongye, Yukui Luo, Xiaolin Xu, and Yunsi Fei. Hammerdodger: a lightweight defense framework against rowhammer attack on dnns. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[11] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*, pages 279–299. Springer, 2016.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[13] Zhezhi He, Adnan Siraj Rakin, Jingtao Li, Chaitali Chakrabarti, and Deliang Fan. Defending and harnessing the bit-flip based adversarial weight attack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14095–14103, 2020.

[14] Fateme S Hosseini, Qi Liu, Fanruo Meng, Chengmo Yang, and Wujie Wen. Safeguarding the intelligence of neural networks with built-in light-weight integrity marks (lima). In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–12. IEEE, 2021.

[15] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. Detection of traffic signs in real-world images: The german traffic sign detection benchmark. In *The 2013 international joint conference on neural networks (IJCNN)*, pages 1–8. Ieee, 2013.

[16] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Mascat: Stopping microarchitectural attacks before execution. *Cryptology ePrint Archive*, 2016.

[17] Mojan Javaheripi, Jung-Woo Chang, and Farinaz Koushanfar. Accchashtag: Accelerated hashing for detecting fault-injection attacks on embedded neural networks. *ACM Journal on Emerging Technologies in Computing Systems*, 19(1):1–20, 2022.

[18] Mojan Javaheripi and Farinaz Koushanfar. Hashtag: Hash signatures for online detection of fault-injection attacks on deep neural networks. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

[19] Biresh Kumar Joardar, Tyler K Bletsch, and Krishnendu Chakrabarty. Machine learning-based rowhammer mitigation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(5):1393–1405, 2022.

[20] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[21] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. {ZebRAM}: Comprehensive and compatible software protection against rowhammer attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 697–710, 2018.

[22] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[23] Jingtao Li, Zhezhi He, Adnan Siraj Rakin, Deliang Fan, and Chaitali Chakrabarti. Neurobfuscator: A full-stack obfuscation tool to mitigate neural architecture stealing. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 248–258. IEEE, 2021.

[24] Jingtao Li, Adnan Siraj Rakin, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. Radar: Run-time adversarial weight attack detection and accuracy recovery. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 790–795. IEEE, 2021.

[25] Jingtao Li, Adnan Siraj Rakin, Yan Xiong, Liangliang Chang, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. Defending bit-flip attack through dnn weight reconstruction. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[26] 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 *Proceedings of the 33th USENIX Security Symposium*, 2024.

[27] Xiang Li, Ying Meng, Junming Chen, Lannan Luo, and Qiang Zeng. {Rowhammer-Based} trojan injection: One bit flip is sufficient for backdooring {DNNs}. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 6319–6337, 2025.

[28] Yu Li, Min Li, Bo Luo, Ye Tian, and Qiang Xu. Deepdyve: Dynamic verification for deep neural networks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 101–112, 2020.

[29] Liang Liu, Yanan Guo, Yueqiang Cheng, Youtao Zhang, and Jun Yang. Generating robust dnn with resistance to bit-flip based adversarial weight attack. *IEEE Transactions on Computers*, 72(2):401–413, 2022.

[30] Qi Liu, Wujie Wen, and Yanzhi Wang. Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2020*, 2020.

[31] Qi Liu, Jieming Yin, Wujie Wen, Chengmo Yang, and Shi Sha. {NeuroPots}: Realtime proactive defense against {Bit-Flip} attacks in neural networks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6347–6364, 2023.

[32] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 131–138. IEEE, 2017.

[33] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlatos, and Baris Kasikci. Siloz: Leveraging dram isolation domains to prevent inter-vm rowhammer. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 417–433, 2023.

[34] Yukui Luo, Adnan Siraj Rakin, Deliang Fan, and Xiaolin Xu. Deepshuffle: A lightweight defense framework against adversarial fault injection attacks on deep neural networks in multi-tenant cloud-fpga. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3293–3310. IEEE, 2024.

[35] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2019.

[36] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1211–1220, 2019.

[37] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Tbt: Targeted neural network attack with bit trojan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13198–13207, 2020.

[38] 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.

[39] Adnan Siraj Rakin, Yukui Luo, Xiaolin Xu, and Deliang Fan. {Deep-Dup}: An adversarial weight duplication attack framework to crush deep neural network in {Multi-Tenant}{FPGA}. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1919–1936, 2021.

[40] Adnan Siraj Rakin, Li Yang, Jingtao Li, Fan Yao, Chaitali Chakrabarti, Yu Cao, Jae-sun Seo, and Deliang Fan. Ra-bnn: Constructing robust & accurate binary neural network to simultaneously defend adversarial bit-flip attack and improve accuracy. *arXiv preprint arXiv:2103.13813*, 2021.

[41] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[42] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

[43] Takami Sato, Junjie Shen, Ningfei Wang, Yunhan Jia, Xue Lin, and Qi Alfred Chen. Dirty road can attack: Security of deep learning based automated lane centering under {Physical-World} attack. In *30th USENIX security symposium (USENIX Security 21)*, pages 3309–3326, 2021.

[44] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[45] Jialai Wang, Ziyuan Zhang, Meiqi Wang, Han Qiu, Tianwei Zhang, Qi Li, Zongpeng Li, Tao Wei, and Chao Zhang. Aegis: Mitigating targeted bit-flip attacks against deep neural networks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2329–2346, 2023.

[46] Xiaobei Yan, Yiming Li, Hao Wang, Han Qiu, and Tianwei Zhang. Bithydra: Towards bit-flip inference cost attack against large language models. *arXiv preprint arXiv:2505.16670*, 2025.

[47] Xiaobei Yan, Xiaoxuan Lou, Guowen Xu, Han Qiu, Shangwei Guo, Chip Hong Chang, and Tianwei Zhang. Mercury: An automated remote side-channel attack to nvidia deep learning accelerator. In *2023 International Conference on Field Programmable Technology (ICFPT)*, pages 188–197, 2023.

[48] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. {DeepHammer}: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1463–1480, 2020.

[49] Ville Yli-Mäyry, Akira Ito, Naofumi Homma, Shivam Bhasin, and Dirmanto Jap. Extraction of binarized neural network architecture and secret parameters using side-channel information. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.

[50] Weiting Zhang, Dong Yang, Wen Wu, Haixia Peng, Ning Zhang, Hongke Zhang, and Xuemin Shen. Optimizing federated learning in distributed industrial iot: A multi-agent approach. *IEEE Journal on Selected Areas in Communications*, 39(12):3688–3703, 2021.

[51] Tong Zhou, Shaolei Ren, and Xiaolin Xu. Obfunas: A neural architecture search-based dnn obfuscation approach. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.