BITSHIELD: Defending Against Bit-Flip Attacks on DNN Executables

Yanzuo Chen[†], Yuanyuan Yuan[†*], Zhibo Liu[†], Sihang Hu[‡], Tianxiang Li[‡], Shuai Wang[†*]
[†]The Hong Kong University of Science and Technology, [‡]Huawei Technologies
[†]{ychenjo,yyuanaq,zliudc,shuaiw}@cse.ust.hk, [‡]{husihang,litianxiang4}@huawei.com

***Abstract*—Recent research has demonstrated the severity and prevalence of bit-flip attacks (BFAs; e.g., with Rowhammer techniques) on deep neural networks (DNNs). BFAs can manipulate DNN prediction and completely deplete DNN intelligence, and can be launched against both DNNs running on deep learning (DL) frameworks like PyTorch, as well as those compiled into standalone executables by DL compilers. While BFA defenses have been proposed for models on DL frameworks, we find them incapable of protecting DNN executables due to the new attack vectors on these executables.**

This paper proposes the first defense against BFA for DNN executables. We first present a motivating study to demonstrate the fragility and unique attack surfaces of DNN executables. Specifically, attackers can flip bits in the `.text` section to alter the computation logic of DNN executables and consequently manipulate DNN predictions; previous defenses guarding model weights can also be easily evaded when implemented in DNN executables. Subsequently, we propose BITSHIELD, a full-fledged defense that detects BFAs targeting both data and `.text` sections in DNN executables. We novelly model BFA on DNN executables as a process to corrupt their semantics, and base BITSHIELD on semantic integrity checks. Moreover, by deliberately fusing code checksum routines into a DNN's semantics, we make BITSHIELD highly resilient against BFAs targeting itself. BITSHIELD is integrated in a popular DL compiler (Amazon TVM) and is compatible with all existing compilation and optimization passes. Unlike prior defenses, BITSHIELD is designed to protect more vulnerable full-precision DNNs and does not assume specific attack methods, exhibiting high generality. BITSHIELD also proactively detects *ongoing* BFA attempts instead of passively hardening DNNs. Evaluations show that BITSHIELD provides strong protection against BFAs (average mitigation rate 97.51%) with low performance overhead (2.47% on average) even when faced with fully white-box, powerful attackers.

I. INTRODUCTION

In recent years, the demand for deep learning (DL) applications in practical settings has escalated, leading to the widespread adoption of deep neural network (DNN) models across a diverse array of computing platforms from cloud servers to embedded devices. To date, an emerging trend is to use DL compilers to compile DNN models from high-level specifications into optimized machine code tailored for various hardware backends [5], [44], [31]. Consequently, rather than being interpreted in frameworks like PyTorch, DNN models can be built into a "standalone" binary format, allowing their direct execution on CPUs, GPUs, or other hardware accelerators. The deployment of DNN executables has notably increased on mobile devicess [33], [17], [35], [32] and cloud computing environments [2], [47].

Existing research has demonstrated that bit-flip attacks (BFAs) enabled by DRAM rowhammer (RH) attacks [23] are effective in manipulating DNN predictions [15], [37], [48] by flipping bits in model weights. While previous works are mostly conducted on DNN models running on DL frameworks like PyTorch, it is unsurprising that DNN executables compiled from DNN models should have similar attack surfaces since they also rely on model weights to make predictions. Importantly, our exploration has identified attack surfaces in the `.text` section of DNN executables, where computation logic (e.g., when to terminate a loop) is implemented.[1] Similar attacks are also noted by recent research in parallel [6]. By flipping bits in the `.text` section, BFA can be conducted more stealthily and effectively without tampering with model weights. This is unique to DNN executables in the standalone formats (details in Sec. II), imposing a new and demanding challenge to secure DNNs in reality.

Despite existing efforts to develop defenses against DNN-targeting BFA [14], [26], [46], [30], [3], [29], [27], they are often limited to quantized model weights and assume specific attack pipelines of BFA, overlooking considerable attack chances (as evaluated and compared in Sec. VIII and Sec. IX). Also, migrating them from highly dynamic DL frameworks to protect DNN executables is often impractical, e.g., Aegis [46] implements a multi-exit DNN structure that generates non-deterministic computation graphs and cannot be compiled into executables. Even worse, previous defenses are designed to only protect model weights, never considering BFA attack surfaces as we identified in the `.text` section of DNN executables. Due to the "standalone" nature of DNN executables, BFAs targeting the `.text` section (i.e., code-based) are *particularly threatening*, as the `.text` section also contains the codebase of the defense mechanism itself (e.g., some checkers [30]). That is, previous defenses are evadable

*Corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2025
24 - 28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.231463
www.ndss-symposium.org

[1]As expected, DNN weights are typically stored in the `.rodata` section of a DNN executable in the ELF format. To ease reading, we refer `.rodata` as "data sections" in the rest of the paper.

by flipping certain bits in their own implementations in the `.text` section (e.g., a `jmp` instruction in a checksum-based defense), as will be shown in Sec. III.

Given the urgency of protecting DNN executables against BFAs and the challenges in doing so, we advocate the following five key requirements for a full-fledged defense. 1) **Generic**: be agnostic to the attack pipeline and types of DNNs; 2) **Unified**: consider both code- and weights-targeting BFAs; 3) **Post-hoc**: do not modify the original DNN or require retraining; 4) **Self-defending**: defend against BFAs that exploit the defense implementation itself (e.g., bypass an integrity check); and 5) **Low overhead**: introduce minimal runtime overhead to ease real-world deployment.

We propose the first defense against BFAs on DNN executables, BITSHIELD, that meets the above five requirements. We model BFA as a process that results in corrupted DNN semantics and propose a novel concept — semantic integrity guard (SIG) — to detect both weights- and code-targeting BFAs by monitoring anomalous decision processes of DNN (Sec. VI). Focusing on the outcome of BFAs, SIG is versatile and does not assume how attackers identify the bits to flip. Moreover, BITSHIELD implements a self-defending mechanism by tightly coupling SIG's computation with code integrity: it uses the code checksum as a "key" to decode ("unmask") correct operands required by SIG's semantic capture process (see Sec. VI-B). Consequently, it is hardly feasible for attackers to induce the intended flips while still producing the correct "key" (code checksum) to silence SIG alarms. BITSHIELD is also a proactive defense: it detects ongoing BFA attempts on-the-fly instead of merely passively hardening the model like many existing works.

BITSHIELD is designed as extra passes during the compilation process, which is agnostic to and does not modify the compiled DNN models. We implement BITSHIELD on top of a production DL compiler, Amazon TVM [5]; that said, BITSHIELD is also portable to other DL compilers with no technical hurdles (see Sec. IX). We evaluate BITSHIELD with nine popular large-scale DNNs and datasets, and consider three fully white-box attacker types with state-of-the-art BFA capabilities: two code-based attackers (with aggressive and stealthy variants) and one weights-based attacker. We also assess BITSHIELD under a range of RH environments and attack surface (e.g., percentage of vulnerable bits with their flip directions) of major DRAM manufacturers. Our main results and contributions are summarized as follows:

- We for the first time mitigate BFAs on DNN executables, a class of rapidly emerging DNN applications yet still exposed to BFA risks. We present a full-fledged defense, BITSHIELD, that considers both weights- and code-based BFAs, is not limited to certain attack pipelines or DNN types, and proactively detects ongoing BFA attempts.
- BITSHIELD is based on the novel concept of semantic integrity to capture anomalous semantics induced by weights- and code-based BFAs. BITSHIELD achieves self-defense by fusing code checksum with DNN semantics. BITSHIELD is post-hoc and can protect pre-trained DNN executables without retraining.
- Evaluations against three adaptive, white-box attackers show that BITSHIELD mitigates 97.51% of active BFA attempts with only 2.47% overhead; BITSHIELD is highly practical for production use.

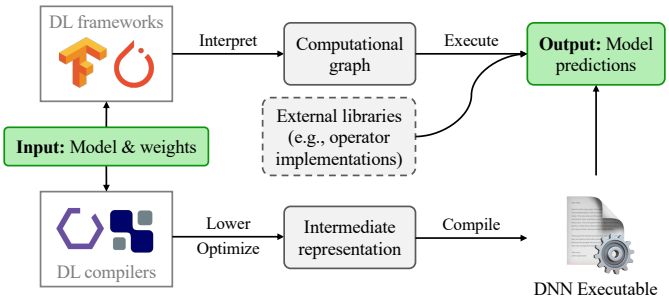


Fig. 1. Comparing DNN models on DL frameworks and as executables.

Artifact: Code and data in this paper are publicly available at https://sites.google.com/view/bitshield-exe.

II. PRELIMINARIES AND RELATED WORK

A. DNN Models and DNN Executables

DNN Models and DL Frameworks. Mathematically, an n-layer DNN model F can be viewed as $f_n \circ f_{n-1} \circ \cdots \circ f_1$, a sequence of progressively propagated non-linear functions f_i $(1 \leq i \leq n)$. Each f_i can be generally formulated as $f_i : y = \mathcal{A}(W, x)$, where x and y are its input and output, respectively. W is the model weights and is formed during training. $\mathcal{A}$ is a composite function of matrix operations. For instance, $\mathcal{A}$ can be a convolution followed by a ReLU activation function, or a simple matrix multiplication, depending on f_i's implementation. Both $\mathcal{A}$ and W can affect the output y when f_i takes x.

As shown in the upper row in Fig. 1, DNN models conventionally run in DL frameworks such as PyTorch and TensorFlow, where the model computational graph is constructed dynamically by interpreting its high-level definition and then executed (possibly after just-in-time compilation) by the framework's runtime. Certain low-level operations (e.g., matrix multiplication and convolution) may also be offloaded to external kernel libraries such as cuDNN [7] on GPUs and MKL-DNN [18] on CPUs.

DNN Executables and DL Compilers. Due to the high demand for deploying DNNs on diverse hardware platforms, DNN models are increasingly compiled into standalone executables that can be directly executed (or invoked by other programs) on target devices. Typically, a DNN executable is self-contained and does not rely on runtimes or offload computation to external libraries; all operators needed by the model are embedded in the `.text` region of the executable, often in specialized, optimized, and fused forms. This is shown in the lower row in Fig. 1.

To compile a high-level DNN model into an executable, modern DL compilers first import its computation graph from

DL frameworks and then parse the graph into an intermediate representation (IR) for optimizations. High-level, platform-agnostic IRs (e.g., Relay IR [43]) are often graph-based, specifying the model's computation flow. Platform-aware IRs (e.g., TVM's TIR) specify how the model is implemented on a specific hardware backend and facilitate hardware-specific optimizations. The final executable is generated by compiling the optimized IR into machine code, often with the help of a backend compiler (e.g., LLVM [25]). Recent research and industry practices have highlighted the real-world usage of DL compilers and DNN executables [5], [44], [31], [19]. The TVM community has received code contributions from companies including Amazon, Facebook, Microsoft, and Qualcomm [9]. DL compilers are increasingly vital to boost DL on CPUs and other heterogeneous hardware backends [2], [47].

B. Bit-Flip Attacks (BFAs)

Rowhammer Attacks. A well-known issue of modern DRAM devices is the disturbance error: repeatedly accessing one DRAM row brings voltage toggling to DRAM word lines, such that the capacitor charge of DRAM cells in the neighboring rows is leaked more quickly. As a result, the memory cell will lose its state (i.e., a bit is flipped) if the remaining charge is insufficient before the next refresh.

Rowhammer attack (RH) [23] exploits the above DRAM disturbance error and empowers recent BFAs. Intuitively, for a target memory cell, RH first carefully identifies a set of neighboring rows as the aggressor rows, and then "hammers" them via frequent row activations. This way, even if attackers do not have direct access to the target bit, they can still flip its value (from 0 to 1 or vice versa). Over the years, researchers have demonstrated the attack on a wide range of memory modules, including DDR3 [23], DDR4 [20], DDR5 [21], and ECC memory [8].

BFAs Towards DNNs. BFAs have been demonstrated as effective in manipulating DNNs. BFAs can deplete DNN intelligence [15], [37], [40], [48] similarly to adversarial examples [12]. However, BFAs are more severe since they can cause the model to "globally" produce undesired outputs, i.e., for *nearly all* inputs instead of just one. BFAs can also be launched to inject backdoors (i.e., targeted BFAs) [4], [38], [39], such that the DNN gives certain predictions for specific inputs while retaining the original predictions for other inputs.

Since exploiting DL frameworks running in Python runtime is challenging due to the dynamic and obscure memory management [6], existing BFAs towards DNNs primarily flip bits in model weights [48], [36]. Given that billions of weight bits may exist and RH attacks are inherently costly, *identifying vulnerable bits* whose flips can effectively affect DNN predictions is critical. For example, DeepHammer [48] localizes bits in active neurons having large gradients, whereas ProFlip [4] searches for salient neurons.

C. Defenses Against DNN BFAs

Existing defenses towards (weights-based) BFAs can be categorized into two groups.

Model Enhancements. One group of defenses aims to improve the resilience of DNN models against BFAs by transforming the model structure or weights. Since quantization can bring better robustness against BFAs targeting model weights, recent works [41], [14] propose binarized DNNs, where model weights and layer outputs are either $+1$ or -1, to significantly increase the difficulty to launch BFAs. Also, to specifically defend targeted BFAs, Aegis [46] proposes a dynamic multi-exit structure, where the DNN's inference terminates early at certain layers for some inputs, such that BFAs flipping bits in subsequent layers will not affect DNN predictions.

While these defenses are effective when training a BFA-resilient DNN from scratch, applying them to a well-trained DNN requires retraining, which is costly in practice and can induce precision drifting between the original and hardened DNNs. Precision drifting (or other unaligned behaviors) incurred by hardening techniques may be likely exploitable, as shown by recent research [34], [10]. Moreover, they are inapplicable to DNN executables: quantization does not notably improve the resilience of these executables, as pointed out by existing works [6] and our study in Sec. III. Aegis, on the other hand, requires dynamic features unsupported by current DL compilers (e.g., adding dynamic exits) and cannot be built into DNN executables.[2]

Weight Integrity. Another line of defense is checking the integrity of model weights at runtime [29], [3], [30], [22]. These schemes often pre-compute a ground truth signature (e.g., a checksum) for the model weights before deployment, and compare it with the signature derived from the weights at runtime. Ideally, since any modifications to the weights can lead to mismatched signatures, these defenses are supposed to defend against all weights-based BFA attempts.

Nevertheless, since DNN executables are standalone binaries, the signature comparison routine, which is implemented as extra functions in the `.text` section, cannot always protect DNN executables as the routine itself can also be attacked by BFAs. E.g., attackers can evade the check by only flipping one bit in the comparison routine, as illustrated in Sec. III.

III. BFA Vectors in DNN Executbales

To motivate our defense design, we first demonstrate the new and pervasive BFA attack vectors in DNN executables. Holistically, given that DL compilers guarantee functional equivalence before and after compilation and do not alter model weight values, one could expect DNN executables to inherit the same BFA vectors (i.e., model weights) as the original high-level models. Our exploration confirms this weight-based BFA vector in DNN executables (see weights-based BFA evaluation in Sec. VIII).

However, further investigation reveals that DNN executables also introduce a new and more severe BFA opportunity, which is unique to DNN executables and is not present in their

[2]We note that Aegis also has a slightly different defense objective of protecting both targeted and untargeted BFAs.

TABLE I
DISTRIBUTION OF VULNERABLE BITS IN OUR SURVEYED DNN EXECUTABLES. ℚ DENOTES QUANTIZED MODELS.

	Model	Dataset	%Acc	#Bits	#Vuln	%Vuln	%Pin
1	ResNet50	CIFAR10	87.69	343248	12070	3.52	72.17
2	ResNet50	MNIST	99.23	343248	13156	3.83	92.82
3	ResNet50	Fashion	88.46	343248	14223	4.14	96.91
4	ResNet50	ImageNet	70.10	459808	22008	4.79	45.30
5	GoogLeNet	CIFAR10	86.15	972440	28926	2.97	74.68
6	GoogLeNet	MNIST	99.23	972440	30401	3.13	80.28
7	GoogLeNet	Fashion	90.00	972440	24381	2.51	82.88
8	DenseNet121	CIFAR10	81.54	1451256	40514	2.79	79.39
9	DenseNet121	MNIST	99.23	1451256	45369	3.13	72.56
10	DenseNet121	Fashion	93.08	1451256	44800	3.09	76.97
11	ℚ-ResNet50	CIFAR10	86.90	728712	15846	2.17	93.61
12	ℚ-GoogLeNet	CIFAR10	84.60	1384904	11588	0.84	91.30
13	ℚ-DenseNet121	CIFAR10	78.50	2666280	13944	0.52	88.69
14	Avg.	-	-	-	-	2.88	80.53

1) #Bits denotes the total number of bits in the `.text` section.
2) #Vuln. is the number of vulnerable bits identified, %Vuln denotes the percentage of vulnerable bits in the total bits, i.e., #Vuln/#Bits.
3) %Pin indicates the percentage vulnerable bits that pins most of the model's output label to a specific one.

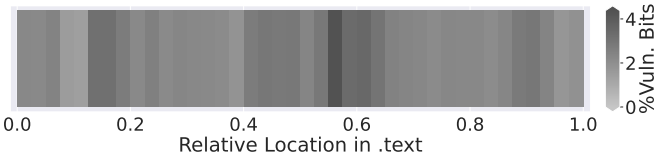
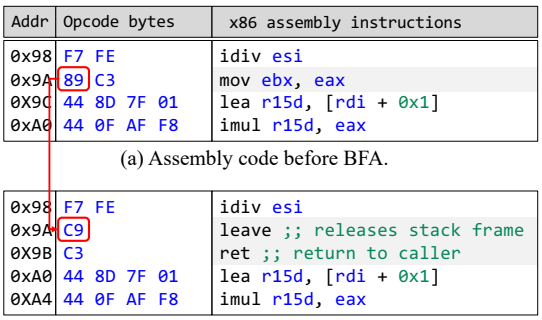


Fig. 2. The distribution heatmap of vulnerable bits in DNN executables.

original high-level models. This new attack vector is code-based BFA, where attackers can flip bits in the `.text` section of the executable to manipulate the model's behavior. We now show how BFA exploits the `.text` section and also evades existing defenses based on integrity checks.

Table I lists our studied DNNs and their accuracy. We pick these models (ResNet50, GoogLeNet, and DenseNet121) due to their representative model structures and wide adoption in various real-world scenarios. We train these DNNs on diverse datasets (CIFAR10, MNIST, Fashion-MNIST, and ImageNet) and compile them into executables using TVM (version 0.9.0), one widely-used DL compiler maintained by Apache Software Foundation. We include all model/dataset combinations with sufficiently high accuracy (above 70%) in our study to ensure a good representation of real-world use cases. We additionally include three cases with quantized models (quantized ResNet50, GoogLeNet, and DenseNet121) to study the effect of quantization on code-based BFAs.

Single-Bit Corruption. We first study, in general, how vulnerable these compiled executables are when faced with code-based BFAs. For each executable, we run a full scan over the bits in its `.text` section and record the model's inference results before and after flipping each bit individually. For this survey, we only consider a bit vulnerable if flipping it causes the model's inference accuracy to drop to the level of a random guess. Later in our evaluation (Sec. VIII), we set the criterion for successful attacks to be an accuracy drop of 3% or more, in order to fully assess our defense. As shown in Table I, we find that each of the studied DNN executables has 11,588 to 45,369 vulnerable bits (on average 2.88% of the bits in the `.text` section), and flipping any of them completely corrupts



(a) Assembly code before BFA.

(b) Assembly code after BFA.

Fig. 3. A code-based BFA example.

the model's inference ability. This illustrates the severity of code-based BFAs, as more than 20 flips are often needed in weights-based BFAs for the same attack goal [48].

Widespread Vulnerable Bits. We also report that the vulnerable bits exist throughout the `.text` section of the DNN executables, as shown in Fig. 2. This corresponds to the observation that vulnerable bits that can be exploited by code-based BFAs are widely distributed in different layers of the model, which is in sharp contrast to the case of weights-based BFAs where the vulnerable bits typically concentrate in the first or last few layers of a model [37], [48]. This also implies that an effective defense against code-based BFAs must not assume the locations of vulnerable bits.

Untargeted & Targeted. Besides depleting DNN intelligence, we also find that targeted BFAs are highly achievable via code-based BFAs. As reported in Table I, a large portion of the vulnerable bits in our surveyed executables (on average 80.53%) can turn most predictions into a certain one (a fixed label), potentially enabling the attacker to manipulate the model's behavior in a targeted manner. This is reasonable as the `.text` section encodes the essential computation procedures of a DNN executable. Consider the flip shown in Fig. 3 where a single bit is flipped at address `0x9A`, causing a `mov` instruction to be replaced with a `leave` instruction followed by a `ret`; the new instructions will unconditionally return to the current function's callee, discarding any succeeding computation. If this flip happens inside a classifier layer, it is possible that the layer's output will just be a chunk of untouched `calloc`-ed memory, leading to an all-zero output and a fixed prediction.

Quantization. By cross-comparing the results of quantized DNNs (11-13th rows) and their floating-point counterparts (1st, 5th, and 8th rows) in Table I, we note that the effectiveness and exploitability of code-based BFAs do not exhibit notable differences in DNN executables with and without quantization. Overall, this is in stark contrast to the case of weights-based attacks, where quantization can significantly reduce the effectiveness of BFAs [37], [48]. Thus, previous defenses by quantizing or binarizing DNNs can hardly defend against code-based BFAs on DNN executables.

Evading Integrity Checks. A serious concern brought by code-based BFAs is their ability to manipulate defenses: due to the standalone nature of DNN executables, these defenses

are also implemented in the `.text` section. Note that this issue is distinct from the "white-box" attacks considered in previous works [30], [46], which exploit defects in a defense's design. In the case of code-based BFA, even theoretically secure defenses such as comparing checksums may be evaded if attackers flip certain critical bits in the checks. Consider Fig. 3 again as a simple example: if there is a checksum comparison after the address `0xA4`, the comparison will always be bypassed after the flip.

IV. Threat Model

DNN Executables. Our defense aims to protect DNN executables compiled by DL compilers. It does not assume any specific types of DNNs and is agnostic to compiler's implementations. Without loss of generality, we consider general floating-point DNNs.

Attacker's Targets. While prior DNN BFA and defense works mainly focus on model weights of DNNs, we argue this is insufficient for DNN executables since attackers can also flip the code bits in the executable's `.text` section; this generally leads to effective and efficient exploitation (see Sec. III). Thus, we assume that attackers can target both the *weights and code bits* in the victim executable, with the end goal of manipulating the victim DNN's behaviors (e.g., downgrading its prediction accuracy or pinning the output to a specific class).

Although DNN's intermediate activations or output vectors can be manipulated by BFA in theory, they are generally *not* the target of DNN-targeting BFA [15], [37], [40], [48]. Recall, as introduced in Sec. II-B, that BFA differs from adversarial example attacks [12] in that it globally modify DNN predictions for nearly all inputs; this requires the flipped bits to "remain flipped" across different inputs. Intermediate activations and output vectors, being frequently updated and overwritten during (every) inference of the DNN, cannot be used to fulfill the persistent requirement. Moreover, manipulating them also requires a specialized attack plan for each DNN input, which downgrades BFA's global attack to a per-input attack. As such, these input-level BFAs are impractical (since launching rowhammer attack is costly) and much less attractive to attackers.

Attacker's Strategy. We assume attackers can flip any desired bit (in data and `.text` sections) as long as it can be flipped by any generic RH techniques using available memory templates. We do not assume specific attack pipeline or profiling strategy (e.g., flipping bits in active neurons or deep layers) of BFAs.

Attacker's Knowledge. We consider severe and representative *white-box attackers* with full knowledge of 1) the DNN executable, including the model structure, weights, and compilation configurations of the executables, and 2) our defense, including the design and implementation details of BitShield. This means that attackers can attempt to evade detection by attacking BitShield itself, e.g., by flipping bits in the defense's implementation in the `.text` section of the executable. Moreover, we assume that attackers can carefully choose the flipped bits so that the DNN executable does not crash during BFA; otherwise, developers and maintainers can easily notice the BFA. We give more details on the three attacker variants in Sec. VIII-B.

Defense Objectives. Consistent with prior works to defend against BFAs [46], [30], [27], [28], we aim to substantially lower the attack success rate on DNNs protected by our defense. Additionally, we aim to provide a proactive defense that can *detect ongoing attacks*, a rather undesired outcome for RH-based attackers who often strive to remain stealthy. From this perspective, we use the terms "detection" and "mitigation" interchangeably in the rest of this paper.

V. Requirements and Considerations

Based on our exploration of prior BFA defenses (Sec. II-C), findings of code-based BFAs on DNN executables (Sec. III), and the strong attacker we assume in Sec. IV, we identify the following key requirements and considerations for a full-fledged defense against BFA on DNN executables.

① ***Generic:*** A defense should be applicable to different DNNs and attack pipelines. Prior defenses often assume BFAs adopt certain strategies to search for bits to flip, which can overlook vulnerabilities if different profiling approaches are used. Other defenses may also be limited to quantized DNNs, leaving floating-point DNNs, which are more prevalent and more fragile to weights-based BFAs, unprotected.

② ***Unified:*** We expect a defense to mitigate both weights- and code-based BFAs. Previous defenses primarily aim at degrading attackers' ability to search for bits to flip, namely they focus on the "source" of BFAs. We argue that this is less practical in defending new BFA vectors (as it requires separately modeling each BFA source), especially given the growing trend of more aggressive optimizations in DNN executables which may expose new BFA chances in the future. In contrast, our defense focuses on the "outcome" of BFA, i.e., whether the model's prediction is manipulated, and characterizes it via the DNN's decision process.

③ ***Post-hoc:*** Users should be able to apply a defense to a pre-trained DNN (executable) in a post-hoc manner. This is vital given the increasing demand of large pre-trained DNNs in specialized tasks. Some previous defenses design new DNN structures to mitigate BFAs, which require retraining model weights, bringing considerable cost. The output DNN also often has inconsistent predictions with the original one, which may introduce new security concerns [10], [34] and also suffer from lower accuracy.

④ ***Self-defending:*** Since code-based BFAs can simultaneously tamper the defense when conducting an attack, a defense needs to be able to protect itself. Here, an attacker not only exploits defects in a defense's design (as we consider a white-box attack), but is also able to modify its low-level implementation. As a result, even those rigorously secure defenses (e.g., checksum comparison) can be easily evaded at negligible costs. Consistent with our defense objectives mentioned above, we do not aim to fully eliminate the chance of such attacks. Rather, we expect to detect an ongoing BFA

that attacks our defense and greatly lower the success rate of bypassing our defense.

⑤ ***Low overhead:*** Considering that DNN executables are often adopted in latency-sensitive applications, it is desired to reduce the overhead of a defense as much as possible. Our defense, with careful optimizations (see Sec. VI-A), is comparable to adding an additional layer into the DNN (a DNN often has dozens or hundreds of layers). Its incurred overhead in practice is negligible and much lower than previous tools, as evaluated in Sec. VIII and Sec. IX.

VI. Design

We first briefly introduce how BitShield meets the five requirements in Sec. V. We then present implementation and optimization details in Sec. VI-A and Sec. VI-B.

Intuitions. Essentially, predictions of a DNN (executable) are made based on a decision process which is jointly decided by the code logic and model weights. In that sense, both code- and weights-based BFAs can be viewed as attacks breaking the normal decision process, regardless of how the attacker identified the bits to flip. Moreover, independent of the attacker's goal (i.e., untargeted or targeted BFA), a successful BFA, when viewed from a specific input's perspective, is reflected as a different and incorrect prediction. The root cause is essentially the anomalous decision process. Therefore, to design a generic (①) and unified (②) BFA defense, the decision process of the protected DNN (executable) should be monitored and checked for anomalies. Similar to "program semantics" formulated in conventional software, this paper deems a DNN's decision process for each input as its *semantics*.

Note that, unlike previous defenses that often assume the attacker's profiling strategy, monitoring the DNN's semantics focuses on the consequence of BFA attacks (without accessing the ground truth prediction) and is therefore agnostic to the attacker's profiling and the attack source. No matter where the BFA is conducted (e.g., towards shallow layers or deep layers, towards weights or `.text`), or how the flipped bits are identified (e.g., via large gradients or salient neurons), it will be detected with high probability as long as it successfully changes the DNN's prediction.

Hardening Workflow. Fig. 4(a) depicts the procedure to compile DNN executables with BitShield. We first insert an initial SIG into the model as additional computational graph nodes to detect BFA-caused anomalous semantics; see the formulation and design in Sec. VI-A. SIG's additional cost is negligible (⑤) considering the number of nodes already present in the graph (e.g., 649 in ResNet50). After the graph is transformed into lower-level IRs, we add self-defense to SIG, which detects BFAs that attempt to corrupt SIG's implementation in the `.text` section (④).

This is done by fusing code checksum into SIG's computation process: as shown in Fig. 4(b), we apply a transformation ("masking") to SIG's operands using the ground truth code checksum obtained at compile time. At runtime, BitShield dynamically applies the inverse transformation ("unmasking") using the actual code checksum calculated on-the-fly (whenever the protected DNN takes an input). If the checksums mismatch, the transformation pair will not cancel out, leading to corrupted SIG computation and thus an alarm; see Sec. VI-B. Moreover, we also ensure that the masking process is implemented with static primitive operations, so they can be optimized and fused with other operators by DL compilers, increasing the work needed for attackers to flip all the desired bits.

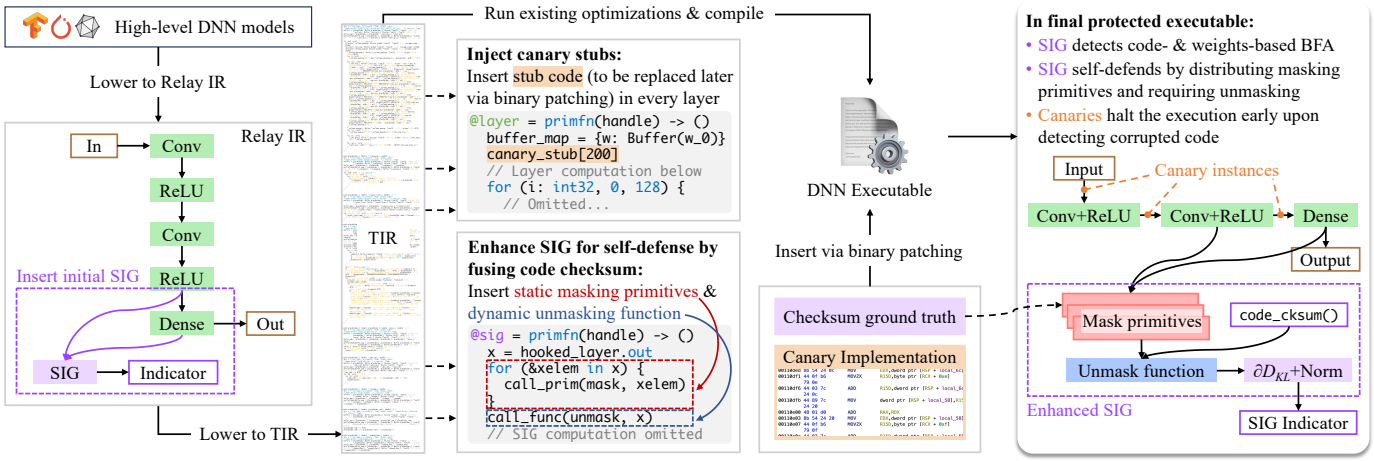
Compiler-Based Design and Application Scope. BitShield is implemented as extra passes in a DL compiler. Overall, the design choice of compiler-based protection ensures that BitShield has maximized generalizability, is compatible with all existing compilation and deployment workflows, and can be applied to protect DNNs of various tasks supported by existing DL compilers. Apparently, this procedure is post-hoc (③) and does not require modifying existing DNN structure or weights, or retraining/fine-tuning the DNN under protection.

Without loss of generality, we focus on classification DNN in the following technical presentation. The classification DNN is also the primary target of existing BFA works [15], [37], [40], [48]. That said, BitShield does not assume any DNN task or implementations; it applies to any DNNs as long as they can be compiled by the de facto DL compilers (e.g., TVM); see discussions in Sec. IX.

A. DNN Semantics and Integrity Guard

Capturing DNN Decisions. As discussed above, no matter how and where the BFA is conducted, it essentially breaks the normal decision process of the DNN, which is deemed as the *semantics* of DNNs. Thus, the first key problem is how to capture the decision process of a DNN when it makes the prediction $y = F(x)$. Here, y is a vector where the i-th element denotes the probability of x belonging to the i-th class. As mentioned in Sec. II-A, a DNN is a composition of multiple non-linear functions; the high non-linearity makes it challenging to represent the decision process. However, inspired by the fact that a randomly initialized DNN's decision process can be trained into a valid one under the guidance of gradients, we can represent this process during runtime by inversely measuring how a model's output can be turned back into a random guess.

To achieve this, we first prepare a vector u that has the same size as y, but with elements all equal (i.e., $= {}^{1}/_{|u|}$), representing equal probability for all classes, or a randomly guessed label. We then compute the distance between u and y. Since both u and y are vectors of probabilities, we follow the common practice [24], [16] and use the Kullback-Leibler (KL) divergence D_{KL} to measure their distance. Finally, similar to conventional back propagation of DNNs, we propagate $D_{KL}(u, y)$ through the last layer f_n to the first layer f_1 and compute the gradients g. The gradients g encode the information for adjusting the DNN's decision process to transform the prediction y back into a random guess u. Formally, let g_i denote the gradients propagated to the i-th layer of weights w_i. We then have



(a) The compilation pipeline with BITSHIELD. (b) Using the protected model for inference.

Fig. 4. An overview of the process of using BITSHIELD to harden DNNs and the execution of the protected DNN executable. SIG is configured to hook into the last two layers in this example.

$$g = \{g_n, \ldots, g_1\},\ g_i = \frac{\partial D_{KL}(u,y)}{\partial f_n(\cdot)} \cdots \frac{\partial f_i(\cdot)}{\partial w_i}. \quad (1)$$

Deciding Normal Semantics. With the above process, we obtain g, gradients representing the "path" from valid predictions to random ones, and rely on this metric to depict the *semantics* of the DNN. We deem an anomalous g a flag of anomalous DNN decision processes — this forms the basis of our semantic integrity guard (SIG). To further decide whether a g is anomalous, we require a reference set G characterizing different normal semantics. Since DNNs are data-driven and their decision logic is formed by the training data, we expect the decision procedures over training data to be normal ones. Thus, for a DNN F, we construct G as the set of all g computed over the training data according to Eq. 1.

Then, whenever the DNN is making a prediction and having the corresponding semantics g^*, we can compare g^* with elements in G to decide whether it is anomalous. The standard way is leveraging probability density to estimate how likely g^* follows the distribution G and accordingly check if g is normal. However, as clarified in Sec. IV, DNN executables are often deployed in latency-sensitive scenarios; estimating probability density is undesirable due to the high overhead. Instead, we can scope the normal "boundary" of G and check if a runtime semantics g^* falls outside the boundary. The boundary can be decided by the maximum and minimum ℓ_1-norm of all $g \in G$. Accordingly, if a user input's g^* falls outside the boundary of G, we consider it anomalous, suggesting ongoing BFA. Note that when viewed from the probability density perspective, g^* lying outside the boundary of G indicates that it is unlikely to follow the distribution of G. In that sense, we view our ℓ_1-norm-based detection as a lightweight approximation of the probability density.

Semantic Integrity Guard (SIG). SIG is implemented as additional nodes on the protected DNN executable's computational graph to calculate gradients. Before deploying a DNN executable, we first insert SIG into the executable. Then, we feed each training input into the executable and compute its semantic g according to Eq. 1. We simultaneously record the minimum, maximum, and average ℓ_1-norm values of all g, denoted as G_{min}, G_{max}, and G_{avg}, respectively.

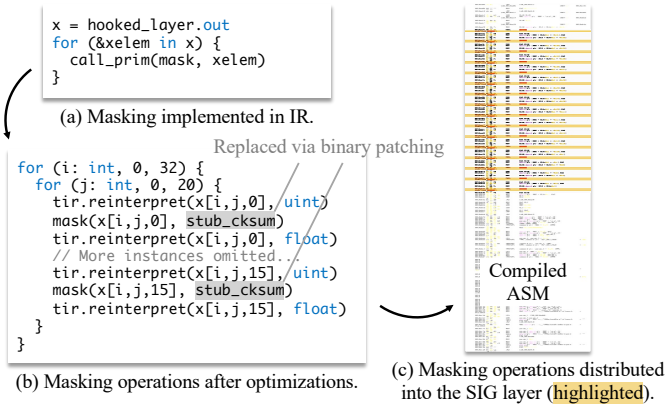
Considering that diverse real-word inputs at runtime may produce semantics whose ℓ_1-norm values fall outside the range $[G_{min}, G_{max}]$, we expand the lower and upper bounds as $L = G_{min} - e \cdot (G_{avg} - G_{min})$ and $U = G_{max} + e \cdot (G_{max} - G_{avg})$, respectively, with e as a configurable factor. We set e to 0.3 in the main experiments. Overall, BITSHIELD's performance is resilient to the choice of e if its value is within a reasonable range (see Sec. VIII-D2). At runtime, when the DNN executable is processing a user input x^*, we deem its semantic g^* normal (the executable is *not* under BFA) only when $L \leq ||g^*||_1 \leq U$; the comparison can be simply implemented by the user.

Optimizations. In practice, we find it often unnecessary to back propagate all layers, whose cost is comparable to an additional inference of the DNN executable. Our tentative explorations show that propagating only the last layer f_n to compute g_n offers sufficient defense capabilities while largely reducing the performance overhead (see our evaluations in Sec. VIII-C). This observation is consistent to empirical evidences in prior work [45], which show that deeper layers in DNNs are more related to the decision process.

B. Achieving Self-Defense

As shown in Sec. III, since DNN executables are standalone, defense mechanisms are implemented in the `.text` section as extra code chunks. This implies that the defense's implementation is also vulnerable to BFAs. The SIG scheme introduced in Sec. VI-A is no exception. Thus, we aim to further introduce another defense mechanism to achieve self-defense, as introduced below.

Vanilla Design. The direct way is to check if contents in the `.text` section is tampered, by pre-computing the code



(a) Masking implemented in IR.

(b) Masking operations after optimizations.

(c) Masking operations distributed into the SIG layer (highlighted).

Fig. 5. The process to distribute the masking operation across the SIG layer.

checksum and comparing it with the runtime code checksum whenever the DNN executable is invoked. However, per our preliminary study and observation in Sec. III, simple checksum comparisons can be trivially evaded by only flipping a few bits in the `.text` section; this only brings negligible extra hurdle to launch a successful BFA.

Fusing Code Checksum with SIG. The above issue appears to be a *chicken-and-egg* dilemma that is hard to address. However, we identify a unique opportunity to solve it by fusing the code checksum into SIG's computation process (and thus the executable's computational graph). The intuition is that, if the code checksum is corrupted, we expect to amplify and propagate the corruption in the DNN's execution process and eventually reflect it via the DNN's semantics.

Without loss of generality, we use $o = W \star v$ as a simplified representation of SIG's computation, where W is the weights, v is a runtime variable (e.g., intermediate gradients), and $\star$ denotes any operation involved in SIG computation. We can rewrite the computation equivalently as $o = \mathcal{M}^{-1}(c^*, \mathcal{M}(c_0, W)) \star v$. Here, $\mathcal{M}$ and $\mathcal{M}^{-1}$ are a pair of weight transformation (*masking*) function and inverse (*unmasking*) function that ensure $\mathcal{M}^{-1}(c^*, \mathcal{M}(c_0, W)) = W$ only when $c^* = c_0$. The c_0 is the ground truth checksum of the `.text` section pre-computed at compile time, and c^* is the `.text` section's checksum calculated at runtime during inference. We choose $\mathcal{M}$ that *significantly* transforms the weights to maximize the chance of a successful detection: in such a case, incorrectly unmasked weights (e.g., due to wrong code checksum caused by bit flips being used) effectively cause an artificial and intended amplification of the ongoing attack, which will likely lead to incorrect computation results in SIG's semantic characterization. In practice, we find XOR operation to be a good choice for $\mathcal{M}$.

Distributing the Transformations. To further combat BFAs that attack the above transformation process, we leverage compiler's optimizations to distribute $\mathcal{M}$'s implementation. Since the masking function $\mathcal{M}$ only takes W and the ground truth code checksum c_0 (obtained at compile time), we implement $\mathcal{M}$ statically using IR primitives. As a result, the compiler's optimization passes will expand $\mathcal{M}$'s implementation into a series of primitive operations *distributed* across the SIG layer. The unmasking operation $\mathcal{M}^{-1}$, in contrast, is implemented as a dynamic tensor-oriented function to support runtime code checksum calculation (see Fig. 4).

We illustrate the distribution process of $\mathcal{M}$ in Fig. 5, where the masking operation is implemented as high-level IR primitives in Fig. 5(a). After the compiler's lowering and optimization passes (e.g., loop unrolling, operator fusion), the IR primitives are expanded into lower-level instructions and fused with other operations wherever possible (Fig. 5(b)), and finally appear distributed across the SIG layer (Fig. 5(c)). This distribution process greatly decreases the probability for attackers to flip all required bits to evade the defense, because any incorrect unmasking propagates the error, and skipping the SIG as a whole also results in corrupted semantics.

Checksum Revisited: A Checksum Canary. In principle, if the DNN executable's code logic (implemented in the `.text` section) is modified or corrupted by BFAs, any subsequent computation will be subject to contingencies and may lead to unexpected results. It is therefore desirable to halt the computation as soon as possible to mitigate the potential damage. In this regard, inconsistent checksums can quickly expose tampered `.text` section, as long as the comparison itself is not attacked. Thus, while we do not (and cannot) directly rely on code checksum comparisons to detect BFAs, we hope to use them as an early stop mechanism upon detecting any corruption in the `.text` section.

We achieve this by leveraging checksum comparison as a canary: we insert checks with pre-computed checksum ground truth values before every layer of the DNN executable, as shown back in Fig. 4(b). Whenever the protected DNN executable runs for inference, the canaries compute the checksum of the executable's `.text` section and compare it with the pre-computed value; if the two mismatch, BitShield *destructs* the DNN execution by inducing an intentional crash, consequently exposing the attack. Specifically, the pre-computed "ground truth" checksum is embedded as a literal into the canaries using standard binary editing techniques immediately after compilation ends. The crash is achieved by issuing an instruction (e.g., the `INT` instruction on x86) to halt the program. We choose Adler-32 [1] as our checksum algorithm for its simplicity and low performance overhead. The same checksum algorithm is also used when fusing code checksum into SIG's computation.

VII. Analysis of Attack Mitigation

Before presenting the empirical evaluations in Sec. VIII, we first provide the following analytical discussions on BitShield as well as the potential BFA threats from various perspectives. Our analysis is depicted in Fig. 6.

As introduced in Sec. VI, BitShield generally delivers a two-stage defense for DNN executables. First, our code checksum canary acts as an early responder and halts the executable if mismatched checksums of the `.text` section is detected. This mitigates BFAs tampering the `.text` section

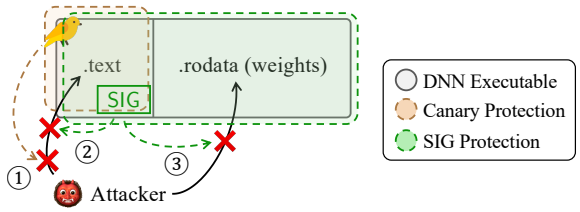


Fig. 6. Different attack types and their mitigation by BITSHIELD components.

(and thus the DNN computation logic); we illustrate the canary protection as ① in Fig. 6.

Adaptive attackers may succeed in evading the checksum comparison by flipping the correct bit(s), e.g., as shown in the example in Sec. III. This is also considered in our design: during the DNN executable's computation, our SIG checks the semantics (i.e., the decision process) of the DNN. Since computations of the SIG is fused with the code checksum via masking and unmasking operations, such an adaptive attack attempt will still be reflected as anomalous semantics and detected by SIG. We show this as ② in Fig. 6.

Weights-based BFA attackers primarily attempt to flip bits in the data sections of the DNN executable, where weights are stored. Overall, if the model weights are tampered, it can also be detected by SIG via anomalous DNN semantics. This is shown as ③ in Fig. 6.

We admit there exists a potential risk that attackers may evade our defense. As mentioned above, we distribute the masking/unmasking operations across a wide range of primitive operations. To evade our defense, a successful BFA must additionally flip bits to 1) bypass the canary, 2) tamper SIG's computation, and 3) manipulate all distributed masking/unmasking operations, where the attacker may also need to adjust all intermediate values already masked with the ground truth code checksum. This significantly increases the difficulty of finding suitable memory templates for RH. Considering the fact that vulnerable bits in DRAM typically exist in a scattered pattern [13], we believe the chance of a successful BFA is extremely low, if not impossible. We further validate this in Sec. VIII using adaptive attackers, and also demonstrate the negligible overhead of BITSHIELD.

VIII. EVALUATION

In this section, we conduct a comprehensive evaluation of BITSHIELD to assess its effectiveness and efficiency. We describe our experimental platform in Sec. VIII-A and introduce the different attacker types we consider in Sec. VIII-B. We then consider three research questions (RQs) in our evaluation:

- **RQ1:** How much performance overhead does BITSHIELD introduce in protected executables?
- **RQ2:** How effective is BITSHIELD in mitigating different types of BFAs?
- **RQ3:** How do different components and parameter configurations of BITSHIELD contribute to its effectiveness?

TABLE II
THE DRAM PLATFORMS WE SIMULATE, WITH EACH PLATFORM'S DEGREE OF VULNERABILITY EXPRESSED AS THE NUMBER OF VULNERABLE BITS PER MILLION BITS OF DRAM. THE PERCENTAGE OF "0→1" FLIPS AMONG ALL FLIPPABLE BITS IS ALSO REPORTED.

Platform Name	A	B	C	D	E
Vuln. bits per 1M bits	0.16	0.39	3.04	26.40	64.54
%"0→1"	51.15	48.89	50.59	50.75	51.16

We answer **RQ1** and **RQ2** in Sec. VIII-C and provide more insights to the results by presenting a comparison with previous defenses in Sec. VIII-E. We answer **RQ3** in Sec. VIII-D.

A. Platform Setup

We first describe how we set up our experimental platform, including the DNN executables we use and the RH environment we simulate. We use a server with an AMD Ryzen Threadripper 3970X 32-core processor and 256GB of RAM running Ubuntu 22.04 for our experiments.

DNN Executables. We use the same executables as in our attack surface survey in Sec. III (excluding executables for quantized models), including 3 popular large-scale DNN models and 3 commonly used datasets. The detailed statistics of the executables are reported in Table I.

RH Environment. We run our attack experiments in a simulated RH environment, which allows us to assess our defense under stronger attackers without being limited by real-world factors like DRAM module types and RH techniques used. To this end, we adopt the vulnerability survey results for DDR4 DRAM from Blacksmith [20], which reports the number of vulnerable bits and their flip directions (i.e., 0→1 or 1→0) for 40 DRAM modules from 4 different manufacturers. This survey is representative of the vulnerability of DDR4 memory modules under the state-of-the-art RH techniques. Prior research also shows that vulnerable bits in DRAM tend to have a uniform distribution [13]. Thus, we configure our simulated RH environment to have different percentages of vulnerable bits matching the survey results, and use these configurations to assess our defense on platforms with different degrees of vulnerability to RH attacks.

Table II shows the DRAM vulnerability data we collected from Blacksmith [20], where A∼D represent the average data from four different manufacturers, Micron, Hynix, Kingston, and Samsung, respectively, and E is the most vulnerable DRAM product among all 40 tested. We use these five platforms as the simulated RH environments in our evaluation. In the Blacksmith survey, the authors sweep over a 256MB region on each DRAM module to find flippable bits. We use the same setting to set up the memory region for our simulated attackers to obtain memory templates; according to our preliminary experiment on real DDR4 DRAM, it takes about 17.5 hours for a real-world attacker to finish the sweep, which we consider reasonably time-consuming. For each platform, we instantiate 50 DRAM modules with different random seeds to repeat the experiments, and average the results for analysis. In reality, RH attackers would also be constrained by the memory templating

and massaging techniques (see Sec. II-B) they employ [48], [42]; we allow arbitrary template reuse and assume 100% success rate for memory massaging in our evaluation to simulate the worst-case scenario for our defense.

B. Attacker Profiles

We consider two variants of code-based attackers and one traditional weights-based attacker in our evaluation. We first describe the goal and capabilities of the attackers, and then introduce each attacker type in detail.

*1) **Attacker Capabilities and Success Criteria**:* All our attackers are fully white-box, meaning they have full knowledge of the victim DNN executable (including the code and weights), the concrete details of our defense mechanism, and the memory templates available on the victim system. We model the launching of each attack as a two-phase process: first, the attacker can perform local profiling using the white-box knowledge of the victim executable, our defense, and the memory templates to generate an attack plan containing a chain of bits to flip. Then, bits in the victim executable are flipped according to the attack plan, and the evaluation metrics are collected. The attackers' end goal is to induce a drop in the inference accuracy of the victim executable without being detected or causing any crash. We follow prior work [30] to set the success criterion: we consider a drop of 3% or more in accuracy as a successful attack.

*2) **Code-Based Attackers**:* We consider two types of code-based attackers: aggressive and stealthy, representing different strategies of selecting the flipped bits. In each code-based BFA experiment, we use either the aggressive or stealthy attacker to generate 50 attack plans for each victim executable; as will be seen in Sec. VIII-C, both attackers launch effective single- and multi-bit attacks, and BITSHIELD is successful in detecting and mitigating them.

Algorithm 1: Attack plan generation for the aggressive code-based attacker (whitebox, adaptive).

Input: Victim executable e, victim host's memory templates $\mathcal{T}$
1 $\mathcal{B} \leftarrow \{\}$
2 **foreach** *bit b in e's* `.text` *section* **do**
3 **if** *flipping b does not trigger* BITSHIELD *& acc. drop* $\geq 3\%$ **then**
4 $\mathcal{B} \leftarrow \mathcal{B} \cup \{b\}$
5 $\mathcal{B} \leftarrow \{b \in \mathcal{B} : \text{flippable}(\mathcal{T}, b)\}$
6 $\mathcal{T}^* \leftarrow \{t \in \mathcal{T} : t$ overlaps with SIG checksums$\}$
7 **if** $\mathcal{T}^*$ *suffices to bypass SIG* **then**
8 **yield** SIG bypass plan
9 $L \leftarrow \text{sorted}(\mathcal{B},$ descending accuracy drop$)$
10 **foreach** *bit b in L* **do**
11 **yield** attack plan $\{b\}$; `// Single-bit attack`
12 **for** $l \leftarrow 2$ **to** $|L|$ **do**
13 $\mathcal{P} \leftarrow \{\}$
14 **foreach** *bit b in L* **do**
15 **if** $|\mathcal{P} \cup \{b\}| = |\{$*instruction of* $b : b \in \mathcal{P} \cup \{b\}\}|$ **then**
16 $\mathcal{P} \leftarrow \mathcal{P} \cup \{b\}$
17 **break if** $|\mathcal{P}| \geq l$ *& acc. drop* $\geq 3\%$
18 **yield** attack plan $\mathcal{P}$; `// Multi-bit attack`

Aggressive Attacker. As mentioned in Sec. III, an attacker can induce bit flips in the `.text` section of a DNN executable to degrade its inference accuracy, frequently to the level of random guesses. This type of attacker represents a malicious actor trying to leverage this observation to launch attacks straightforwardly. We show the attack plan generation algorithm for the aggressive code-based attacker in Alg. 1. During local profiling (lines 1–6), the attacker iterates over all bits in the victim executable's `.text` section and records the impact of flipping each bit on the accuracy of the executable.

To simulate a fully whitebox attacker trying to bypass our defense, bits that are unflippable on the victim host due to the lack of memory templates, or bits that will trigger our defense if flipped, are not recorded or used in attacks. Also, the attacker searches for memory templates that may be used to tamper the ground truth checksum values embedded in SIG. In case the attacker gathers enough templates to modify the embedded checksums at will, a flipping plan to bypass SIG can be generated to increase the attack success rate (line 8). To create plans to achieve the objective of accuracy degradation, the recorded bits are then sorted by their impact on the accuracy (line 9). We then first generate single-bit attack plans (lines 10–11), where each plan contains a single bit to flip from the sorted list. Multi-bit attack plans are also generated (lines 12–18) to evaluate BITSHIELD's robustness against more complex attacks: each plan contains the most effective (aggressive) bit(s) from the list such that the attack goal is reached while the number of flips and crash probability are minimized.

Stealthy Attacker. A stealthy attacker aims to decrease the victim model's accuracy without causing too large accuracy drops: instead of trying to flip the most effective vulnerable bit(s) in an attack, this attacker tries to flip less effective ones, as long as the criterion of a 3% accuracy drop for a successful attack is met. We propose a straightforward way to simulate this attacker: similar to the aggressive attacker in Alg. 1, the attacker scans the victim executable's `.text` section during local profiling to record vulnerable bits. For an attack plan, however, the attacker selects the least effective (stealthy) bit(s) among those causing at least a 3% accuracy drop, instead of the most effective ones.

*3) **Weights-Based Attacker**:* For the weights-based attacker, we simulate a strong adversary proposed by Rakin et al. [37]; we use the default values for the parameters in this attacker. During local profiling, the attacker conducts a gradient-based vulnerable bit search consisting of three steps: in-layer search, cross-layer search, and progressive iteration. In an in-layer search for a DNN model f, the n_b (by default 1) most vulnerable bits in a layer are found by calculating the gradient of the loss function $\mathcal{L}$ w.r.t. all the bits $\mathcal{B}$ in the layer's weights and selecting the n_b bits with the largest absolute gradient values: $\hat{\mathcal{B}} = \text{Top}_{n_b}|\nabla_{\mathcal{B}}\mathcal{L}(f(x), y)|$, where x is the input, y is the ground-truth label, and Top_{n_b} returns the n_b bits with the largest absolute gradient values. Every bit $\hat{b} \in \hat{\mathcal{B}}$ is then flipped by XORing it with a mask to maximize the total loss.

Once the new loss after the flips is calculated and recorded, they are undone (for later steps). The in-layer step is repeated for all layers in the DNN to form a cross-layer search, and only the n_b most vulnerable bits across all layers are actually

TABLE III
PERFORMANCE OVERHEAD OF BITSHIELD BY COMPARING UNPROTECTED (VANILLA) AND PROTECTED EXECUTABLES.

Model		Vanilla (ms)	Protected (ms)	Overhead (%)
ResNet50	CIFAR10	21.07	21.63	2.66
	MNIST	21.39	21.79	1.87
	Fashion	21.39	21.90	2.38
	ImageNet	33.21	35.94	8.22
	Avg.	**24.27**	**25.32**	**4.33**
GoogLeNet	CIFAR10	55.59	56.13	0.97
	MNIST	55.90	56.14	0.43
	Fashion	55.93	56.29	0.64
	Avg.	**55.81**	**56.19**	**0.68**
DenseNet121	CIFAR10	22.10	22.71	2.76
	MNIST	22.07	22.64	2.58
	Fashion	22.11	22.60	2.22
	Avg.	**22.09**	**22.65**	**2.52**
Avg.		-	-	**2.47**

flipped at the end. The attacker can increment n_b and retry after the cross-layer search until the attack goal or maximum number of iterations (by default 20) is reached. The set of bits selected to be flipped at the end of the iteration denotes the attack plan for the victim executable.

Note that we consider a powerful white-box attacker who tries to evade our defense with full knowledge of BITSHIELD's mechanisms. Hence, the weights-based attacker avoids flipping the bits which have the most influence on the SIG output (as determined by their gradients). Unlike prior defenses assuming a fixed percentage of unflipped bits [30], our attacker is more stealthy by dynamically increasing the fraction of evaded bits (i.e., unflipped to avoid being detected) in each search iteration if the attack is detected, until reaching a maximum fraction; we evaluate BITSHIELD towards different maximum evasion fractions in Sec. VIII-C3. Other variants of weights-based BFAs, e.g., T-BFA [39] where the Rowhammer-specific instance [48] is restricted to flipping one bit per page, are omitted as they represent more constrained attackers.

C. RQ1&2: Defense Results of BITSHIELD

*1) **Performance Overhead**:* We measure the performance overhead of BITSHIELD on the protected executables by recording the inference time of each executable using TVM's built-in `benchmark` utility. Results are reported in Table III, where the "Vanilla" and "Protected" columns show the inference time of an executable before and after being protected by BITSHIELD, respectively, and the "Overhead" column shows the percentage increase in inference time with protection. Our defense brings a very low performance overhead to the DNN executables, with an average overhead of 2.47%; indicating that BITSHIELD is suitable for deployment even in performance-critical production systems.

We observe no obvious difference between the overheads of different models when trained on the same dataset. Specifically, in the cases of ResNet50 and DenseNet121, BITSHIELD shows similar levels of overhead (2.30% vs. 2.52%), despite DenseNet121 having double the number of layers compared to ResNet50 and a 4.23× larger `.text` region. This indicates that BITSHIELD scales well with complex models and large executables. On the other hand, ResNet50 trained on ImageNet has the highest overhead of all cases (8.22%), primarily due to its larger last layer as a result of increased number of classes, which produces more computational workload for SIG. However, when compared to cases with other datasets, we find that a 100× increase in the number of classes only brings a 3.57× higher overhead, and the overhead is still low in its absolute value. This case demonstrates that BITSHIELD is also scalable with the complexity of datasets with a reasonable scaling coefficient.

*2) **Attacking Vanilla Executables**:* We evaluate the effectiveness of our defense against the attackers described in Sec. VIII-B by measuring their attack success rate (ASR), as well as the average accuracy drop induced by those successful attacks. We first show the attack results on vanilla (unprotected) executables averaged over all simulated DRAM platforms in Table IV. For all the unprotected vanilla executables, all attackers achieve high ASRs (98.75% on average). However, we notice that code-based attackers have higher ASRs than weights-based attackers, indicating that the former may have looser requirements for available memory templates and pose a more serious threat to DNN executables; this also underlines the importance of defending against code-based attackers on these compiled models.

In terms of accuracy drops, weights-based and aggressive code-based attackers both cause large accuracy drops, greatly reducing the executables' inference capabilities. The stealthy code-based attacker, on the other hand, causes smaller accuracy drops (as designed in Sec. VIII-B2); the high ASR of the aggressive and stealthy attackers shows the possibility of inducing varying degrees of damage to the victim executable via code-based BFA.

*3) **Attacking Protected Executables**:* Table V shows the attack results on executables protected by BITSHIELD, where the ASRs are reduced to 2.49% on average.[3] We observe that BITSHIELD successfully mitigate all attacks from code-based attackers (including single- and multi-bit attacks) and 92.52% of attacks from the weights-based attacker on average. Given that all the attackers are adaptive adversaries actively trying to bypass our defense (Sec. VIII-B), we consider this a strong and encouraging result. We also confirm that, in all experiment runs on all platforms, no attacker is able to find sufficient memory templates ($\mathcal{T}^*$ in Alg. 1) to tamper SIG's distributed masking/unmasking operations. On the other hand, we notice that even if the weights-based attacker is successful, the accuracy drop caused by the attack is significantly reduced (only dropping to 54.10% on average) and the attacker can no longer degrade the model to a random guesser. We provide the comparison with previous defenses in Sec. VIII-E.

Following Sec. VIII-B3, we also evaluate BITSHIELD w.r.t. different evasion fractions set by the weights-based attacker. Results are plotted in Fig. 7, where the average values for the metrics (lines) as well as their 95% confidence intervals (shaded regions) are shown. We observe that, when the

[3]To understand the ASR rate, we conduct a standard hypothesis test to determine the confidence level of the result. We report that at the ASR of 2.49%, BITSHIELD can mitigate 91.5% of attacks with 99% confidence.

TABLE IV
ATTACK RESULTS ON VANILLA DNN EXECUTABLES WITHOUT PROTECTION.

Attack Success Rate (%)

Attacker Type	ResNet50				GoogLeNet			DenseNet121			Avg.
	CIFAR10	MNIST	Fashion	ImageNet	CIFAR10	MNIST	Fashion	CIFAR10	MNIST	Fashion	
Aggressive code-based	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Stealthy code-based	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Weights-based	98.80	91.60	94.00	96.00	98.80	92.40	96.80	98.00	97.20	98.80	96.24
Avg.	99.60	97.20	98.00	98.67	99.60	97.47	98.93	99.33	99.07	99.60	98.75

Accuracy after Attack (%)

Attacker Type	ResNet50				GoogLeNet			DenseNet121			Avg.
	CIFAR10	MNIST	Fashion	ImageNet	CIFAR10	MNIST	Fashion	CIFAR10	MNIST	Fashion	
Aggressive code-based	18.09	13.85	15.31	2.59	12.11	11.80	12.61	11.98	13.31	15.26	12.69
Stealthy code-based	82.17	89.90	78.54	63.46	72.30	82.44	83.75	74.19	91.83	85.14	80.37
Weights-based	18.26	11.07	10.17	2.94	12.95	10.28	10.45	10.79	11.40	11.17	10.95

TABLE V
ATTACK RESULTS ON DNN EXECUTABLES PROTECTED BY BITSHIELD.

Attack Success Rate (%)

Attacker Type	ResNet50				GoogLeNet			DenseNet121			Avg.
	CIFAR10	MNIST	Fashion	ImageNet	CIFAR10	MNIST	Fashion	CIFAR10	MNIST	Fashion	
Aggressive code-based	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Stealthy code-based	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Weights-based	16.00	1.20	3.20	6.40	24.80	1.60	1.20	2.80	12.80	4.80	7.48
Avg.	5.33	0.40	1.07	2.13	8.27	0.53	0.40	0.93	4.27	1.60	2.49

Accuracy after Attack (%)

Attacker Type	ResNet50				GoogLeNet			DenseNet121			Avg.
	CIFAR10	MNIST	Fashion	ImageNet	CIFAR10	MNIST	Fashion	CIFAR10	MNIST	Fashion	
Aggressive code-based	-	-	-	-	-	-	-	-	-	-	-
Stealthy code-based	-	-	-	-	-	-	-	-	-	-	-
Weights-based	66.35	45.64	31.25	51.54	74.84	90.00	68.72	40.00	37.31	35.38	54.10

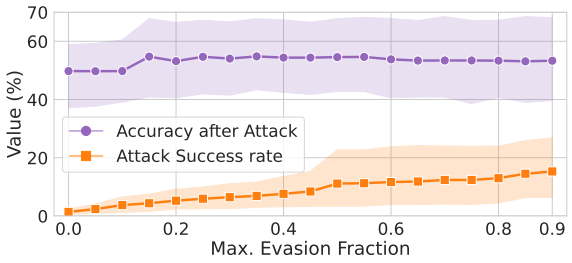


Fig. 7. The attack success rate and model accuracy after attack under different maximum evasion fraction settings for the weights-based attacker.

TABLE VI
BREAKDOWN OF THE ATTACK OUTCOMES ON PROTECTED RESNET50(RN), GOOGLENET(GN), AND DENSENET121(DN).

Attacker	Outcome	Models			Sum (Proportion)
		RN	GN	DN	
Code-based	Profiling failed	422	1062	862	2346 (31.28%)
	SIG	386	342	526	1254 (16.72%)
	Canary	1049	96	112	1257 (16.76%)
	Accuracy	143	0	0	143 (1.91%)
	Success	0	0	0	0 (0%)
Weights-based	Profiling failed	49	24	15	88 (1.17%)
	SIG	884	657	684	2225 (29.67%)
	Accuracy	0	0	0	0 (0%)
	Success	67	69	51	187 (2.49%)
Sum		3000	2250	2250	7500 (100.00%)

weights-based attacker does not try to evade detection (in which case it declines to a non-adaptive attacker) or only tries to evade a small fraction of SIG-affecting weight bits, BITSHIELD can effectively mitigate almost all the attacks (with an attack success rate close to 0%).

On the other hand, when the evasion fraction is set to 0.9, the average attack success rate raises to 15.29%, with its 95% confidence interval also slightly expanded (where its upper bound is still under 30%). Additionally, we observe that the inference accuracy remains quite consistent across all evasion settings after attack, with just minor fluctuations. It tends to stabilize at approximately the average value of 53.28%. This indicates that it is substantially difficult for adaptive weights-based attackers to completely deplete the inference intelligence of a model protected by BITSHIELD.

D. RQ3: Ablation Studies

*1) **Defense Components**:* To study the effectiveness of BITSHIELD's various defense components, we show in Table VI a breakdown of the attack outcomes of all three attackers on the protected executables. Here, "Success" indicates the number of successful attacks, and the other labels show the number of failed attacks due to different root causes: "Profiling failed" means that the adaptive attacker fails to find attackable vulnerable bits during the local profiling phase (Sec. VIII-B1), possibly due to lack of memory templates usable under the defense. "SIG" and "Canary" (i.e., our code checksum canary) mark the attacks detected by the corresponding defense components, respectively. "Accuracy" stands for cases where the attacker's flipping plan does not cause enough accuracy drop compared to their objective, and can be the case if the attacker falls back to worse attack plans due to the defense.

For both weights- and code-based attackers, BITSHIELD increases the difficulty for them to search for attackable bits on protected DNN executables, as indicated by failed attacks falling under the "Profiling failed" category. For attackers that manage to find attackable bits locally, we find that BITSHIELD is able to detect most of the attacks at runtime as well, as shown by the number of failed attacks detected by the

TABLE VII
EFFECTS OF DIFFERENT e VALUES.

e	Model	FA (%)	MF (%)	ΔASR (%)		
				Code-based	Weights-based	Avg.
0.0	ResNet50	0.00	6.93	0.00	0.00	0.00
	GoogLeNet	0.20	6.16	0.00	0.00	0.00
0.3	ResNet50	0.00	1.32	-	-	-
	GoogLeNet	0.00	0.01	-	-	-
0.4	ResNet50	0.00	0.81	0.00	0.00	0.00
	GoogLeNet	0.00	0.00	0.00	0.00	0.00
0.5	ResNet50	0.00	0.37	0.00	0.00	0.00
	GoogLeNet	0.00	0.00	0.00	0.00	0.00
0.6	ResNet50	0.00	0.27	0.00	0.00	0.00
	GoogLeNet	0.00	0.00	0.00	0.00	0.00
1.0	ResNet50	0.00	0.00	0.00	0.00	0.00
	GoogLeNet	0.00	0.00	0.00	+8.00	+4.00

1) ResNet50 and GoogLeNet are trained on CIFAR10 and MNIST datasets.
2) FA: false alarm of test inputs, MF: mis-flag of inputs from other datasets.
3) ΔASR: changed ASR w.r.t. $e = 0.3$ in main experiments.

defense components. One illustrative example is the ResNet50 case, where there are relatively more successful local profiling attempts to find attackable bits than in the others, but the majority of these attacks are still detected by BITSHIELD at runtime, resulting still in a low number of successful attacks (67 out of 3,000).

In addition, as described in Sec. VI, our code checksum canary acts as an "early responder" to attacks while SIG is the "catch-all" even if the canary is bypassed by the attacker in an earlier stage (e.g., due to being skipped, as mentioned in Sec. III). This is consistent with the observation from Table VI that a larger number of attacks are detected by SIG when compared to the canary. In all cases, we notice that both defense components are involved and necessary to effectively detect BFAs, which shows the effectiveness of our multi-component design.

*2) **Normal Datasets and** e **in SIG**:* As mentioned in Sec. VI-A, SIG provides a user-configurable parameter e to expand the captured normal semantic range derived from G; in our main experiments, we set it to a predefined value 0.3. In this section, we run additional experiments to explore the effect of different values for e on the performance of BITSHIELD. We use ResNet50 trained on CIFAR10 and GoogLeNet trained on MNIST as the models and set e to varying values from 0.0 to 1.0. The RH environment is set to platform E from Table II, representing the most vulnerable DRAM module.

We report in Table VII the false alarm (FA) rate by feeding the protected model the test split of the dataset it is trained on and checking if SIG raises any alarm. We find $e = 0.0$ to be a too low value, as it raises the FA rate to 0.2% for GoogLeNet; other cases have no FA, meaning that legitimate users are never affected. The ΔASR is evaluated by running the same white-box attackers from Sec. VIII-B and recording how different BFAs' ASRs change w.r.t. $e = 0.3$ in the main experiments. We observe that the ΔASR is 0 (no difference) in most cases, indicating that varying e does not significantly impact the attack detection capability of SIG, but too large values (i.e., the last row in Table VII, where $e = 1.0$) may result in overlooked attacks.

TABLE VIII
COMPARISON WITH PRIOR DEFENSES ON ADAPTIVE WEIGHTS-BASED ATTACKS. ONLY WEIGHTS-BASED ATTACKS ARE CONSIDERED, AS NONE OF THE PREVIOUS METHODS PROTECT AGAINST CODE-BASED BFAS.

Work	Method	Performance overhead (%)	Acc. loss (%)	Mitigation rate (%)
Aegis [46]	Enhance structure	NA (< 0)	1.24	63.76
DeepAttest [3]	Fingerprint	7.20	≤ 0.09	90.00
NeuroPots [30]	Enhance weights + fingerprint	3.93	1.38	100.00
Ours	Semantic integrity	2.47	NA (0)	92.52

Moreover, since SIG leverages the protected DNN's training data to decide normal semantics, we also evaluate SIG's reaction to (legitimate) inputs that are dissimilar to the training data: we run CIFAR10 models on MNIST inputs and vice versa, plus 10 classes of images from ImageNet [11]. While such cross-dataset inputs indicate *inappropriate* usages of the DNN, we use them to assess SIG's resilience to more diverse real-world inputs. The mis-flag rates (MF) w.r.t. varied e are reported in Table VII. We observe that SIG flags very few of these inputs, and the MF rate is close to 0 except for $e = 0.0$. We however note that these flags do *not* cause performance overhead or model crashes, as the code in `.text` is intact and its checksum is unchanged. We conclude that 0.3∼0.6 is a suitable range for e in most cases with 0.3 being a reasonable default; users can adjust it to fit their specific needs.

E. Comparison with Previous Defenses

In this section, BITSHIELD is compared with recent, state-of-the-art defenses employing different design principles. Since all existing defenses are designed for weights-based attacks and cannot provide protection against code-based ones (and also may not be implementable on DNN executables), we compare BITSHIELD with them on weights-based attacks. We aim to provide more insights into the performance and effectiveness of various methods in this field.

As mentioned in Sec. II-C, existing defenses generally fall into two categories: (1) new or enhanced DNN implementations (fine-tuning model weights or transforming model structures for better BFA robustness), e.g., Aegis [46] converts the protected model into a dynamic multi-exit DNN; and (2) weight integrity monitoring (deriving a checksum value from the model weights and comparing it with the ground truth), e.g., DeepAttest [3] which calculates a fingerprint from the monitored model weights. Note that while DeepAttest also fine-tunes the model weights, it requires this step to embed the fingerprint into the model weights, not for improving the model's BFA robustness. Finally, NeuroPots [30] represents a new, hybrid approach where weight enhancements and weight fingerprinting are combined. We thus compare BITSHIELD with these three defenses. Note that the weights-based attackers here are also adaptive adversaries attempting to evade the defenses.

Our comparison is based on the performance overhead, accuracy loss (brought to the protected model by the defense), and the mitigation rate against adaptive attackers with defense evasion capabilities. Results are presented in Table VIII. We report that DeepAttest, NeuroPots, and BITSHIELD have sim-

ilar mitigation rates of weights-based attacks, with NeuroPots being slightly higher than the others. In terms of runtime overhead, BITSHIELD is the lowest among all defenses with non-negative overheads; Aegis reports a negative overhead on average because many inputs are able to exit the inference process early after its multi-exit transformation, but the generality of this result may not be guaranteed. As all three methods other than ours require modifications to the protected model weights, they also slightly degrade the model's accuracy, where NeuroPots brings an average accuracy loss of 1.38%. Our method does not cause accuracy loss because it is post-hoc and does not modify model weights.

IX. DISCUSSION

Extensibility. We discuss the extensibility of BITSHIELD from various perspectives. First, the current evaluation of this paper only considers DNNs for classification tasks. We clarify that current DL compilers primarily support these DNNs and have incomplete support for other DNNs, e.g., those for natural language processing (NLP) which involve recurrent neural networks (RNNs). Given that said, our SIG design should be compatible with any DNN where gradients can be computed (i.e., if they are trainable), and our canary operates at a lower level where the DNN model has been compiled into a binary. Thus, extending BITSHIELD to other types of DNN should not be a concern.

Second, we implement BITSHIELD on TVM because it has relatively complete support for instrumentation and extension APIs. However, our design is generic and has no TVM-specific dependencies; Glow [44], for example, also has its IR for us to perform computational graph transformations, but it is not mature and does not (yet) expose sufficient interfaces for extension. Overall, compiling and accelerating DNN models with hardware primitives is a very active research area; we believe that BITSHIELD is portable to other systems with no extra research challenge.

Besides, careful readers may question that there may exist alternative or more expert strategies for the adaptive attackers to implement. Nevertheless, this is a fundamentally challenging problem even for real-world attackers. Considering that, for instance, past work attacking an executable's code bits [13] only found vulnerable bits through manual analysis but we use an automated process (Sec. III), and that our adaptive attacker successfully generates many attack plans partially bypassing BITSHIELD's components (Table VI), we believe that our adaptive attackers are sufficiently strong in practice.

DoS Attacks via Adversarial Inputs. While BFAs aim to manipulate DNN predictions for legitimate inputs, knowledgeable readers may wonder if carefully crafted adversarial inputs can be leveraged to launch DoS attacks on BITSHIELD (e.g., making BITSHIELD crash the protected executable constantly). We clarify that such "DoS opportunity" does *not* exist in BITSHIELD, as the canary module is only evoked when the `.text` section is tampered, which is not possible with adversarial inputs. Although adversarial inputs may be crafted to trigger false alarms in SIG, these attempts, if at all possible, only quickly expose the presence of the attacker without being able to cause crashes or interrupt other normal users.

X. CONCLUSION

We propose the first general and unified BFA defense on DNN executables, BITSHIELD, based on DNN semantics. BITSHIELD is post-hoc, self-defending, and efficient. Evaluation on large-scale DNNs and diverse datasets against fully white-box, adaptive attackers shows that BITSHIELD provides strong BFA protection with low costs.

ACKNOWLEDGEMENT

The HKUST authors were supported in part by an NSFC/RGC JRS grant under the contract N_HKUST605/23 and an RGC CRF grant under the contract C6015-23G.

REFERENCES

[1] zlib – a massively spiffy yet delicately unobtrusive compression library. https://www.zlib.net/.

[2] Amazon. Amazon SageMaker Neo uses Apache TVM for performance improvement on hardware target. https://aws.amazon.com/sagemaker/neo/, 2021.

[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 *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 487–498.

[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 OSDI*, 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] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[8] 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.

[9] TVM Community. Tvm deep learning compiler joins apache software foundation. https://tvm.apache.org/2019/03/18/tvm-apache-announcement, 2019.

[10] Edoardo Debenedetti, Giorgio Severi, Nicholas Carlini, Christopher A Choquette-Choo, Matthew Jagielski, Milad Nasr, Eric Wallace, and Florian Tramèr. Privacy side channels in machine learning systems. *USENIX Security*, 2024.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR*, pages 248–255. IEEE, 2009.

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

[13] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261.

[14] 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)*, pages 14083–14091. IEEE.

[15] Sanghyun Hong, Pietro Frigo, Yigitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *USENIX Security Symposium*, pages 497–514, 2019.

[16] Rui Huang, Andrew Geng, and Yixuan Li. On the Importance of Gradients for Detecting Distributional Shifts in the Wild. In *Advances in Neural Information Processing Systems*, volume 34, pages 677–689. Curran Associates, Inc.

[17] Texas Instruments. The AM335x microprocessors support TVM. https://software-dl.ti.com/processor-sdk-linux/esd/docs/latest/linux/Foundational_Components/Machine_Learning/tvm.html, 2021.

[18] Intel. MKL-DNN for scalable deep learning. https://software.intel.com/en-us/articles/introducing-dnn-primitives-in-intelr-mkl, 2017.

[19] Animesh Jain, Shoubhik Bhattacharya, Masahiro Masuda, Vin Sharma, and Yida Wang. Efficient execution of quantized deep learning models: A compiler approach. *arXiv preprint arXiv:2006.10226*, 2020.

[20] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 716–734. IEEE, 2022.

[21] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. {ZenHammer}: Rowhammer Attacks on {AMD} Zen-based Platforms. pages 1615–1633.

[22] 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.

[23] 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.

[24] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[25] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[26] 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.

[27] 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*, CCS '20, pages 101–112. Association for Computing Machinery.

[28] Liang Liu, Yanan Guo, Yueqiang Cheng, Youtao Zhang, and Jun Yang. Generating Robust DNN With Resistance to Bit-Flip Based Adversarial Weight Attack. 72(2):401–413.

[29] Qi Liu, Wujie Wen, and Yanzhi Wang. Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–8. ACM.

[30] Qi Liu, Jieming Yin, Wujie Wen, Chengmo Yang, and Shi Sha. {NeuroPots}: Realtime Proactive Defense against {Bit-Flip} Attacks in Neural Networks. pages 6347–6364.

[31] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX OSDI*, pages 881–897, 2020.

[34] Qi Pang, Yuanyuan Yuan, and Shuai Wang. MPCDiff: Testing and repairing mpc-hardened deep learning models.. In *NDSS*, 2024.

[32] NXP. NXP uses Glow to optimize models for low-power NXP MCUs. https://www.nxp.com/company/blog/glow-compiler-optimizes-neural-networks-for-low-power-nxp-mcus:BL-OPTIMIZES-NEURAL-NETWORKS, 2020.

[33] OctoML. OctoML leverages TVM to optimize and deploy models. https://octoml.ai/features/maximize-performance/, 2021.

[35] Qualcomm. Qualcomm contributes Hexagon DSP improvements to the Apache TVM community. https://developer.qualcomm.com/blog/tvm-open-source-compiler-now-includes-initial-support-qualcomm-hexagon-dsp, 2020.

[36] 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)*, pages 1211–1220. IEEE.

[37] 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.

[38] 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.

[39] 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.

[40] 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*, 2021.

[41] 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.

[42] Kaveh Razavi, Ben Gras, Cristiano Giuffrida, Erik Bosman, Bart Preneel, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. page 19.

[43] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Logan Weber, Josh Pollock, Luis Vega, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. Relay: A high-level compiler for deep learning. *arXiv preprint arXiv:1904.08368*, 2019.

[44] 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*, 2018.

[45] Philip Sperl, Ching-Yu Kao, Peng Chen, Xiao Lei, and Konstantin Böttinger. DLA: Dense-Layer-Analysis for Adversarial Example Detection. pages 198–215. IEEE Computer Society.

[46] 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 *USENIX Security 2023*.

[47] Sally Ward-Foxton. Google and Nvidia Tie in MLPerf; Graphcore and Habana Debut. https://www.eetimes.com/google-and-nvidia-tie-in-mlperf-graphcore-and-habana-debut/#, 2021.

[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.