Compiled Models, Built-In Exploits: Uncovering Pervasive Bit-Flip Attack Surfaces in DNN Executables

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

***Abstract*—Recent research has shown that bit-flip attacks (BFAs) can manipulate deep neural networks (DNNs) via DRAM Rowhammer exploitations. For high-level DNN models running on deep learning (DL) frameworks like PyTorch, extensive BFAs have been conducted to flip bits in model weights and shown effective. Defenses have also been proposed to guard model weights. Nevertheless, DNNs are increasingly compiled into DNN executables by DL compilers to leverage hardware primitives. These executables manifest new and distinct computation paradigms; we find existing research failing to accurately capture and expose the attack surface of BFAs on DNN executables.**

To this end, we launch the first systematic study of BFAs on DNN executables and reveal new attack surfaces neglected or underestimated in previous work. Specifically, prior BFAs in DL frameworks are limited to attacking model weights and assume a strong whitebox attacker with full knowledge of victim model weights, which is unrealistic as weights are often confidential. In contrast, we find that BFAs on DNN executables can achieve high effectiveness by exploiting the model structure (usually stored in the executable code), which only requires knowing the (often public) model structure. Importantly, such structure-based BFAs are pervasive, transferable, and more severe (e.g., single-bit flips lead to successful attacks) in DNN executables; they also slip past existing defenses.

To realistically demonstrate the new attack surfaces, we assume a weak and more realistic attacker with no knowledge of victim model weights. We design an automated tool to identify vulnerable bits in victim executables with high confidence (70% compared to the baseline 2%). Launching this tool on DDR4 DRAM, we show that only 1.4 flips on average are needed to fully downgrade the accuracy of victim executables, including quantized models which could require 23× more flips previously, to random guesses. We comprehensively evaluate 16 DNN executables, covering three large-scale DNN models trained on three commonly-used datasets compiled by the two most popular DL compilers. Our finding calls for incorporating security mechanisms in future DNN compilation toolchains.

I. Introduction

Recent years have witnessed increasing demand for applications of deep learning (DL) in real-world scenarios. This demand has led to extensive deployment of deep neural network (DNN) models in a wide spectrum of computing platforms, ranging from cloud servers to embedded devices. To date, a promising trend is to use DL compilers to compile DNN models in high-level model specifications into optimized machine code for a variety of hardware backends [15], [75], [54]. Hence, instead of being interpreted in frameworks like PyTorch, DNN models can be shipped in a "standalone" binary format and executed directly on CPUs, GPUs, or other hardware accelerators. More and more DNN executables have been deployed on mobile devices [60], [37], [66], [59] and cloud computing scenarios [4], [86].

Despite the prosperous adoption of DNN executables in real-world scenarios, their attack surface is largely unexplored. In particular, existing research has demonstrated that bit-flip attacks (BFAs) enabled by DRAM Rowhammer (RH) are effective in manipulating DNN models [35], [69]; defenses have also been proposed accordingly. However, existing attacks and defenses only apply to BFAs on DNN models in DL frameworks like PyTorch, leading to a greatly underestimated attack surface of BFA on DNN executables because: (1) Prior works mostly attack and protect the weights in a DNN model, neglecting the fact that the model structure becomes more readily attackable in compiled, standalone executables. (2) As earlier attacks frequently need the gradients of victim model weights, a strong, whitebox attacker with full knowledge of the victim model is usually an assumed requirement, which may not be realistic given that model weights and training data are often confidential. On the contrary, attackers leveraging point (1) *do not* need whitebox knowledge, as we will explain in this paper. (3) Existing defenses, being designed specifically for weights-based BFAs, fail to consider or provide protection against attacks on DNN executables, resulting in a false sense of security. Thus, it is high time that a systematic study on the attack surface of BFAs on DNN executables be conducted. To this end, our work provides the first and in-depth understanding of the severity of BFAs on DNN executables. Our findings suggest the need to incorporate comprehensive mechanisms in DNN compilation toolchains to harden real-world DNN executables against exploitations.

Importantly, among the vulnerable bits in model structures, we find that a large portion of them are transferable between DNN executables sharing the same model structure

[*]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.2025.230419
www.ndss-symposium.org

(see Sec. VI-D and Sec. VI-E). We design a novel vulnerable bit searcher based on this observation that can be used by attackers to identify vulnerable bits in the victim executable with high confidence, increasing the search success rate from 2% (baseline) to 70%. This supersedes the strong, whitebox attacker requirement in previous works and shows that BFAs can be launched by attackers knowing only the victim model structure which is often public or recoverable [8], [89], [36]. Using our search tool, we demonstrate that only 1.4 flips on average are needed to completely destroy the inference capability of a DNN model, including quantized models which have been considered more robust and required 2× to 23× more bit flips than their full-precision versions previously [88], [35]. We also adopt and augment an RH attack technique [40] to show successful exploitations on DNN executables deployed on real-world DDR4 devices.

Our extensive study is conducted over DNN executables emitted by two commonly used production-level DL compilers, TVM [15] and Glow [75], developed by Amazon and Meta (Facebook), respectively. We assess the attack surface on diverse combinations of DNN models, datasets, and compilers, covered by a total of 16 DNN executables in this study. We made important observations, including ❶ *Pervasiveness*: we identify on average 16,599 vulnerable bits in each of the DNN executables we studied, even for quantized models, which have been known to be more robust than full-precision models [88], [35]. ❷ *Effectiveness & Stealthiness*: 71.1% of the RH attacks reported in this work succeed with only a single bit flip, while 95.6% succeed within 3 flips, making RH attacks highly effective in practice. ❸ *Versatility*: we show that BFAs can achieve various attack end-goals over both classification and generative models. ❹ *Transferability*: we also find many "superbits" — vulnerable bits that exist across DNN executables sharing the same model structure (`.text` section) but with different weights. To demonstrate the feasibility of ❺ *Practical exploitation*, we launch our attack on DDR4 DRAM modules and show that attackers succeed with just 1.4 flips on average, meaning that the high cost for techniques like RH is no longer an obstacle. We further conduct reverse engineering and manual analysis to characterize those vulnerable bits. Our work highlights the need to incorporate security mechanisms in future DNN compilation toolchains to enhance the reliability of real-world DNN executables against exploitations. In summary, we contribute the following:

- This paper launches the first in-depth study on the attack surface of DNN executables under BFA, revealing the underestimated severity of BFA on DNN models compiled as executables.
- We instantiate our observations as an RH-based attack to show how BFAs on DNN executables can be launched under a threat model more realistic and restrictive for attackers than before. We design a novel method to identify vulnerable bits in the victim executable with ∼70% accuracy (compared to ∼2% in the baseline) and assess our attack on DDR4 DRAM modules.
- Our empirical findings uncover the pervasiveness, stealthiness, versatility, and transferability of BFA vectors on DNN executables. We present case studies and root cause analysis to characterize the vulnerable bits in DNN executables.
- We release our attack artifact, including all scripts and data, to the research community at https://sites.google.com/view/exe-single-bit-bfa [1].

II. Preliminaries and Motivations

A. DL Compilers and DNN Executables

DNN Compilation. DL compilers typically accept a high-level description of a *well-trained* DNN model, exported from DL frameworks like PyTorch, as their input. During compilation, DL compilers often convert the model into intermediate representations (IRs) for optimizations. High-level, platform-agnostic IRs are often graph-based, specifying the model's computation flow. Platform-specific IRs such as TVM's TensorIR and Glow's High Level Optimizer (HLO) specify how the DNN model is implemented on a specific hardware backend and support hardware-specific optimizations. Optimizations performed by DL compilers often include constant folding, operator fusion (e.g., fusing a ReLU operator with a preceding convolution operator), platform-aware scheduling, and others. Finally, DL compilers convert their low-level IRs into assembly code (or first into standard LLVM/CUDA IR [44], [58]).

DNN Executables. Popular DL compilers including TVM [15] and Glow [75] emit DNN executables in the standard ELF format to be executed on mainstream CPUs and other hardware. The emitted DNN executables can be in a standalone executable or a shared library (a `.so` file loadable by other programs). Without loss of generality, we focus on the `.so` format in this paper; our attack pipeline and findings can be easily applied to standalone executables. Traditionally, DL frameworks essentially interpret the DNN model as a computational graph and offload low-level computation to external kernel libraries like cuDNN [17] and MKL-DNN [38]. DNN executables, in contrast, usually do not rely on runtime libraries, but have all computation operations compiled into the binary (e.g., via just-in-time compilation), often in specialized form and fused with other operations.

Real-World Usage. The real-world usage of DL compilers and DNN executables has been illustrated in recent research [15], [75], [54], [39] and industry practice. The TVM community has reported that TVM has received code contributions from companies including Amazon, Facebook (Meta), Microsoft, and Qualcomm [19]. While GPUs are often used for DNN tasks today, DL compilers fulfill the emerging demand for a wide range of other platforms. TVM has been used to compile DNN models for CPUs [51], [39]. Facebook has deployed Glow-compiled DNN models on CPUs [57]. Overall, DL compilers are increasingly vital to boost DL on CPUs, embedded devices, and other heterogeneous hardware backends [4], [86]. This work exploits the output of DL compilers, i.e., DNN executables, and we, for the first time, show the pervasiveness and severity of BFA vectors in DNN executables.

B. Bit-Flip Attacks

BFA via Rowhammer Attack. BFA is a type of hardware fault injection attack that corrupts the memory content of a target system (by flipping its bits). While BFA can be initialized via a variety of hardware faults [21], RH [42] manifests as one highly effective, practical, and controlled fault injection attack. In short, RH exploits DRAM disturbance errors, such that for some modern mainstream DRAM devices, repeatedly accessing a row of memory can cause bit flips in adjacent rows. RH roots in the fact that frequent accesses on one DRAM row introduce voltage toggling on DRAM word lines. This results in quicker leakage of capacitor charge for DRAM cells in the neighboring rows. If sufficient charge is leaked before the next scheduled refresh, the memory cell will eventually lose its state, and a *bit flip* is induced. By carefully selecting neighboring rows (aggressor rows) and intentionally performing frequent row activations ("hammering"), attackers can manipulate some bits without directly accessing them. For DDR3 DRAM, RH attacks are easily launched by alternating accesses to the two rows adjacent to the victim row ("double-sided hammering"). For DDR4, more recent research has pointed out that attackers need to precisely control the frequency-domain parameters (phase, frequency, and amplitude) of the hammering procedure to induce bit flips [40]. While contemporary BFA research is mostly demonstrated on DDR3 DRAM devices [43], [88], [68], this work demonstrates RH attacks on recent DDR4 DRAM devices.

BFA on DNN. Recent research has specifically launched BFA toward DNN models [14], [35], [69], [70], [71], [72], [88]. The attack goals of BFAs can be generally divided into two categories: first, BFA may be launched to extensively manipulate model predictions over all inputs, possibly reducing the model accuracy to random guesses. In contrast to typical DNN training that aims to minimize the loss, BFA strives to maximize the loss function $\mathcal{L}$ to temper inputs [69], [88]. Meanwhile, targeted BFA (T-BFA) aims to manipulate prediction output over specific inputs [71]. T-BFA retains the original predictions over the other inputs to offer a stealthier attack. For both types of attacks, existing works primarily launch BFA to flip bits in the model weights. For example, ProFlip [14] identifies and flips the weights of specific salient neurons to extremely large values, which can control the overall predictions for certain classes. On the other hand, different defense mechanisms have also been proposed to protect DNN models from BFAs. Recent research has shown that models are more robust against BFAs after being quantized, requiring 2× to 23× more bit flips to achieve the same attack goals compared to their floating-point counterparts [88], [69], [35]. In this work, however, we show that quantization does *not* grant DNN executables more robustness against BFAs, and attackers can still succeed by flipping a single bit, as shown in Sec. VI-E. In fact, most existing defenses cannot be ported to DNN executables or cannot protect against attacks targeting these executables, as we will discuss more in Sec. VII-A.

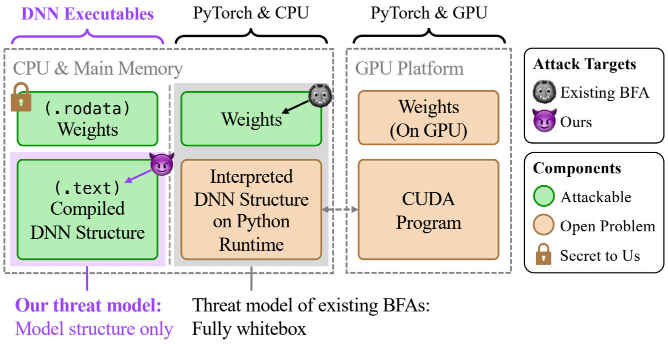


Fig. 1. Runtime systems of DNN models in DL frameworks and in executables. Here, "secret" means they are technically attackable as demonstrated in prior works [71], [88], [85], but in reality they are often *unknown* to attackers.

C. Research Motivation

BFA Specialized for DNN executables. Fig. 1 compares deploying DNN executables in CPU & main memory with running DNN models in PyTorch. Existing BFAs on DL frameworks (e.g., [35], [69]) primarily target the trained weights of DNN models while using their model structures to compute gradients. On the other hand, launching hardware exploitations against interpretation-based environments like the Python runtime is an open problem and only few works have presented limited demonstrations [64]. It is also unclear whether GPU memories are vulnerable to BFAs. On DNN executables, however, the model structure is also statically compiled into the binary (in contrast to being dynamically interpreted on high-level frameworks), making it more readily attackable. And compared to code-targeting BFAs on non-DNN programs [26] which can hardly be automated due to the complexity of generic programs, attackers can exploit DNN's task-specific nature to derive metrics and oracles to automate the search for vulnerable bits, rendering the attack more scalable and practical.[1]

Thus, this research analyzes the attack vectors in DNN executables, particularly their `.text` sections that contain the model structure information (indicated by "our attack target" in Fig. 1, as opposed to existing BFA which targets model weights). In essence, previous weights-based BFAs break the data flow integrity while our structure-based BFAs attack both the control and data flow (see case studies in Sec. VI-F) of the victim model; this type of BFA also exposes the gap in BFA defenses on DNN executables, as mentioned in Sec. II-B.

BFA on DNN Executables with Weak Attackers. To locate vulnerable bits in model weights, prior works often use gradient-based searching and thus require model weights to be known to the attacker (i.e., a fully whitebox attacker is needed). We instead assume a weak, graybox attacker who

[1] Also for this reason, it is not suitable to compare BFAs on DNNs to those on non-DNN programs, as it may result in unaligned attack objectives and potentially questionable results.

only knows the model structure without any information about the weights (as marked by the lock in Fig. 1 on `.rodata` which usually stores the weights); we consider this a more realistic scenario as model weights are often proprietary but the model structure is often public or recoverable [8], [89], [36]. Under this assumption, we show the feasibility of launching effective BFAs on DNN executables by leveraging our findings on DNN executables' BFA surfaces. We also design an automated search tool to identify vulnerable bits in the victim executable with high confidence (see Sec. IV).

III. Threat Model and Assumptions

Attacker's Target and Goals. In this paper, attackers launch BFAs targeting the model structure (in the `.text` section, as shown in Fig. 1) of victim DNN executables. The attacker has two end goals for the two representative types of DNN models in this paper, respectively: for *classifiers*, we successfully downgrade the inference accuracy of the victim DNN model to random guess, and for *generative models*, we temper the generation results to get biased or distorted outputs. As clarified in Sec. IV-B and empirically shown in Sec. VI-C, our downgrading attack can be easily extended to more targeted attacks (often referred to as T-BFA), e.g., manipulating the classification outputs of specific inputs to a target class. In addition, tempered generation results can consequently result in model poisoning attacks when used in data augmentation [11], [55], [77].

While DNN's outputs may also be deceived by certain crafted inputs (e.g., adversarial examples [25]), they generally tamper the feature extraction process (i.e., *algorithmic* vulnerabilities) and manipulate the output for *each* crafted input. In contrast, our attack and other BFAs exploit the *implementation* vulnerabilities of DNNs, making them malfunction when processing almost *all* normal, legitimate inputs.

Environment. Our attack targets DNN executables compiled by DL compilers, deployed on a resource-sharing machine-learning-as-a-service (MLaaS) environment [74]. The attacker is co-located with the victim and can run an unprivileged user process on the same machine as the victim DNN executable. The attack happens after the victim executable is loaded into the memory and ready for execution, as is consistent with existing works. We assume that proper isolation mechanisms are in place to prevent the attacker from directly accessing any victim-owned files or memory pages. The attacker launches BFA using currently mature RH exploitation techniques, whose steps include memory templating [73], memory massaging [76], [68], and hammering [40], [24]. We also align with other BFAs like [88] to minimize the number of flips for an attack, considering the high cost of launching RH in practice. Our assumptions are reasonable and, in fact, represent a weaker attacker than prior attacks [71], [88], [85].

During attacks, no software-level vulnerabilities on the victim DNN executables or the host machine is required, and the attacker does not feed maliciously crafted inputs ("adversarial examples" [25]) to the victim DNN executable. The victim DNN executable exposes its public query interface (e.g., for normal users to submit medical images and obtain diagnosis); the attacker can submit benign inputs to get the model's outputs via this public interface during RH.

TABLE I

A comparison of our threat model with related works, where ✔ and ✕ signify "required" and "unneeded," respectively.

	Stealing Attacks [68], [31]	**DeepHammer** [88]	**T-BFA** [71]	**Ours**
Model Structure	✔	✔	✔	✔
Weights	✕	✔	✔	✕
Training Data	✔	✕	✕	✕
Victim File Readable	✕	✔	✔	✕
Common Shared Library	✔	✕	✕	✕
Attacker Goal	Duplicate DNN's Functionality	Manipulate DNN's Outputs		

Knowledge of the Victim DNN Model. We assume that only the victim model's structure is known to the attacker. This is a practical assumption because model structures (in contrast to weights) are often public.[2] Even in the case of private or partially known model structures, recent works have demonstrated the feasibility of recovering the full details of DNN structures [36], [93], [87], [53], [52]; attackers can infer the model structure with these techniques before launching our attack. With the model structure, the attacker can construct and compile a set of same-structure-different-weights models (Sec. IV-C) to facilitate offline bit searching (Sec. IV-B), which will allow her to attack the victim DNN even when the victim's weights are completely unknown.

Comparison with Existing Works. We compare our threat model with the most recent related works in Table I, where we assume the weakest attacker, needing only the victim DNN's model structure to manipulate its outputs.

All prior relevant BFAs [5], [14], [70], [71], [88] assume that attackers have full knowledge of the victim DNN model including the model structure and the trained weights to, e.g., compute gradients. We deem this as an overly strong assumption: DNN weights are generally trained on private data and viewed as the key intellectual property of DNNs. In practice, only DNN owners have access to the trained weights, and no existing attacks can fully recover DNN weights.[3]

While RH and similar hardware fault injection techniques are employed to steal DNN's model weights in recent works [68], [31] (a different attack objective, as indicated in the 2nd column in Table I), they require a portion of the victim's labeled training dataset [68], [31]; we have no such requirements because training data is often, if not always, more confidential than the trained weights. Moreover, their recovered weights do *not* reduce the requirement for victim's weight knowledge in weight-targeting BFAs, because they are only functionally similar to the actual weights and cannot aid the attacker's gradient computation.

[2] Most commercial DNNs are built on public well-defined backbones, e.g., Transformer [84] is the building block of the GPT models [10].

[3] Query-based model extraction [81], [61] only obtain DNNs *functionally similar* to the victim DNN but does not recover the *exact* weight values.

Our attack does not need (even read-only) access to the victim's files as in prior works [88], [71], and thanks to the compact nature of DNN executables, we can further drop the requirement of common shared libraries between the attacker and victim [68]; see our attack pipeline in Sec. IV-B. Overall, our requirements are quite permissive for the attacker; it is indeed a looser set of assumptions when compared to prior techniques launching BFAs toward DNN models in high-level DL frameworks, which work in a purely whitebox scenario.

IV. Assault from a Weak Attacker

A. *Overview and "Superbits"*

Under the assumption of a weak attacker (Sec. III), we present our attack pipeline in Sec. IV-B to show how DNN executables can be attacked. Importantly, our attacker benefits from transferable *superbits* — vulnerable bits that exist across DNN executables (compiled using the same DL compiler) sharing the same model structure but with different weights, as mentioned in Sec. I. These superbits are key in allowing the attacker to launch the attack without whitebox knowledge of the victim DNN.

Aligned with previous BFAs towards DNNs [88], [71], our attacker starts with a local profiling step where she constructs DNN executables with the same structure as the victim executable. This enables her to conduct an offline bit search (in an attacker-controlled, simulated environment) and identify these superbits. We present in Sec. IV-C the rationale and method of constructing the profiling DNN executables. Once the bit search finishes, the online attack is conducted via RH exploitation in the real-world environment; we demonstrate the attack on DDR4 DRAM in Sec. VI-E.

B. *BFA Pipeline*

Criteria for Vulnerable Bits. We first give the attacker's definition for "vulnerable bits" for classification and generative DNNs, respectively. For classifier DNNs, vulnerable bits are bits that cause the inference accuracy of the model to drop to random guess ($\approx \frac{1}{\text{\#classes}}$) once flipped. For a generative model, vulnerable bits cause its output image quality or semantics to drastically change (see details in Sec. VI-A). In either case, vulnerable bits *cannot* cause the DNN to crash. Moreover, since this paper focuses on *single-bit* BFAs (unlike prior works which need a chain of bit flips to achieve the same effect [88], [69]), we do not consider vulnerable bits that need to be chained with others. Fig. 2 depicts our attack pipeline which consists of three steps:

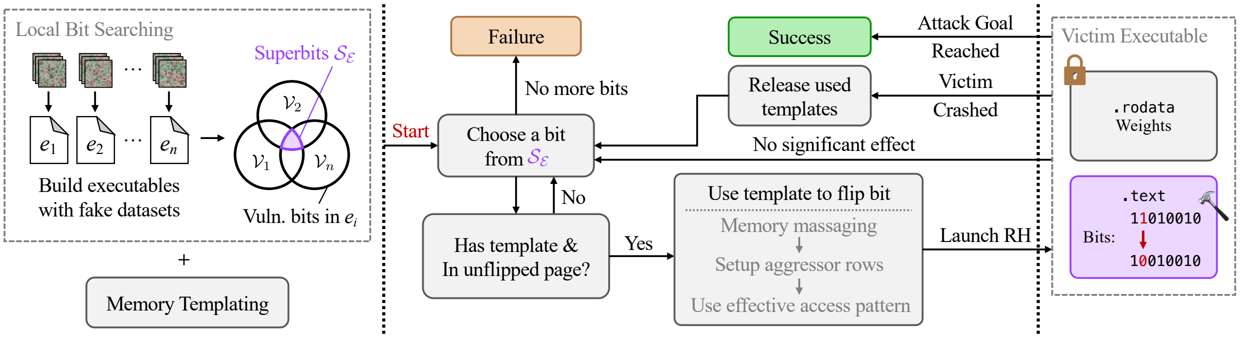
Offline: Searching for "Superbits." As mentioned in Sec. III, we assume the attacker to have no knowledge about the weights or training data of the victim model; she must rely on what she knows (the model structure) to decide which bits in the victim executable's `.text` region to attempt to flip. We achieve this with a novel offline bit search method. Our evaluation shows that the attacker can use this approach to confidently identify such superbits in the victim executable with ∼70% accuracy, compared to ∼2% in the baseline case where the attacker randomly selects bits to flip, as will be shown in Sec. VI-E. We now give more details on the method.

Let f_θ be the victim DNN model, where θ denotes its weights and f its structure. Let e be the executable obtained by compiling f_θ. First, the attacker locally prepares a series of n DNN executables $\mathcal{E} = \{e_1, e_2, ..., e_n\}$, where each $e_i \in \mathcal{E}$ shares the identical structure with the victim DNN e but with different weights. Here, we require that weights in each e_i be well-trained (noted as "trained weights"). That is, the weights must be the output of a training process with an optimizer, instead of being randomly initialized. We elaborate on the rationale and how to obtain them in Sec. IV-C.

After obtaining $\mathcal{E}$, the attacker starts to use them as local profiling targets. Ultimately, our goal is to find vulnerable bits in e that can be flipped to cause a desired effect, but this is challenging because it is normally hard to tell whether an arbitrary bit will be a vulnerable bit in the victim executable e whose weights are never exposed to the attacker. Recall that, as mentioned earlier, we identify superbits that are transferable among a set $\mathcal{E}$ of DNN executables with distinct weights; attackers can leverage this transferability to search for vulnerable bits. More specifically, the problem of searching for vulnerable bits in the victim e can be transformed into finding superbits shared by $\mathcal{E}$ (denoted by $\mathcal{S}_\mathcal{E}$) that are likely also shared by e; we give details below.

Given a set of n DNN executables $\mathcal{E} = \{e_1, e_2, ..., e_n\}$, we define the superbits among $\mathcal{E}$ as the vulnerable bits shared by all of them: $\mathcal{S}_\mathcal{E} \overset{\text{def}}{=} \bigcap_{i=1}^{n} \mathcal{V}_i$, where $\mathcal{V}_i$ is the set of vulnerable bits found in e_i, as illustrated in Fig. 2(a). As the size of $\mathcal{E}$ increases, $\mathcal{S}_\mathcal{E}$ becomes a set of superbits that are shared by more and more e_i's. Intuitively, these superbits are also more likely to affect the victim e as well, since e has the same model structure as e_i. Although the superbits' vulnerability in e is not guaranteed and the attacker will not know until she launches the online attack towards e, our empirical results show that this approach finds vulnerable bits in e with high enough accuracy for real-world attackers to conveniently launch practical attacks, as we will demonstrate in Sec. VI-E.

To obtain $\mathcal{V}_i$ for every e_i, a naïve attacker would need to sweep all bits in each e_i's `.text` section and check if the vulnerability condition is met, then repeat this process for all $e_i \in \mathcal{E}$. However, this can be very time-consuming, especially for complex DNNs that can have more than 6.8 million bits in their `.text` sections, according to our preliminary study. To speed up the search, the sweeping process and intersection calculation can be interleaved so that the attacker can iteratively shrink $\mathcal{S}_\mathcal{E}$, starting with $\mathcal{S}_\mathcal{E} = \mathcal{V}_1$, each iteration trying the bits already in $\mathcal{S}_\mathcal{E}$ on an unswept binary e' and removing those that are not vulnerable in e', instead of trying all bits in e'. During this process, the attacker will need to flip bits in the local executables $\mathcal{E}$ and assess their effects. We clarify that this is simple: as the attacker has full control over $\mathcal{E}$ and her local environment, she can easily achieve bit flipping by editing the binary files.



(a) Preparation phase. (b) Online RH-based attack in an MLaaS environment.

Fig. 2. The attack pipeline. The "lock" symbol in the top right means attackers cannot access the weights of the victim.

Once the offline bit search finishes, the attacker can proceed to the online steps. While we present an RH-based attack, our bit search algorithm is generic and can be used with other BFA techniques as well, such as using laser beams [9].

Online Preparation: Memory Templating. Our attack is achievable in practical settings: we consider the attacker to have no direct access to victim-owned resources and no knowledge on victim's weights. As a "warm up," the attacker needs to scan the DRAM module in the host machine for bit locations that can be flipped using RH, a procedure called memory templating [73]. A bit is flippable using RH only if there is a "template" in the DRAM module with the same bit location and flip direction. For DDR4 platforms, an effective "DRAM access pattern [40]" containing the frequency-domain parameters needed to launch RH on the platform must also be found to trigger a flip. The set of metadata describing where RH can flip bits and how to flip each bit is called *memory templates*. Currently, multiple tools have been made available to find these templates on DDR4, including Blacksmith [40] and TRRespass [24]. In our pipeline, we leverage and slightly extend Blacksmith, the state-of-the-art RH technique for DDR4, to perform memory templating and use the access patterns it found in the attack step later.

Online Attack: RH. After determining the set of superbits $\mathcal{S}_{\mathcal{E}}$ and obtaining the memory templates, the attacker can launch RH to flip the vulnerable bits in e. She first chooses a superbit $s \in \mathcal{S}_{\mathcal{E}}$ that has at least one available template, whose information has been obtained in the "warm-up" phase. Then, as shown in Fig. 2(b), she also needs to check if the bit belongs to an unflipped page: due to limitations of current RH techniques, as pointed by prior BFA works [73], [88], we restrict our attacker to be able to flip only one bit per physical page, unless the victim executable crashed and restarted, in which case its allocated memory is regarded as reset. If all these requirements are met, the attacker can then use the template to flip the bit via standard RH, whose steps include memory massaging, setting up aggressor rows, and applying the effective access pattern. Here, memory massaging refers to the process of precisely placing the memory page containing the bit to flip at the location specified by the template [76], [73], [88], [43], [82], [68]. More specifically, the attacker can abuse the per-CPU page frame cache in Linux to highly precisely relocate the victim pages to vulnerable locations [43], [68], [88], and given the usually small code page sizes in DNN executables ($\ll$ 2MB; see Sec. V), this can be done without the assistance of additional side channel attacks or common shared libraries between the attacker and victim [68].

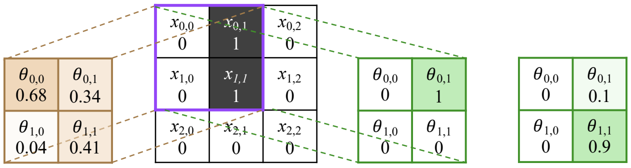
Once the bit flip is triggered by RH, the attacker queries the victim executable via its public interface to check if its behavior has changed as expected. As shown in Fig. 2(b), there are three possible outcomes: (1) the victim's behavior changes as expected, which is the desired outcome; (2) the victim crashes, which is undesirable since attackers wish to manipulate the victim's behavior without crashing it; and (3) the victim's behavior does not change, which is also undesirable. In the latter two cases, the attacker has to repeat the above process with a different superbit s until she finds a bit that can be flipped and changes the victim's behavior as expected, or runs out of possible superbits to try. Our experiments on DDR4 DRAM (Sec. VI-E) show that the attacker succeeds on the first flip attempt 71.1% of the time, and succeeds within 3 attempts 95.6% of the time. Moreover, our attack is agnostic to RH techniques, and the above steps can replaced by any other RH techniques if available [42], [27], [48]. Our attack is also not coupled with the DDR4 standard; DDR3 or DDR5 platforms can be targeted as well, given that the attacker picks a suitable RH technique.

C. Preparing DNN Executables with Well-Trained Weights for Local Profiling

We mentioned in Sec. IV-B that attackers need trained weights to construct the local set of DNN executables $\mathcal{E}$. We now elaborate on the rationale and challenges for getting them, as well as how we efficiently and effectively obtain enough of them for a relatively large $\mathcal{E}$ (e.g., when $|\mathcal{E}| > 10$).

DNN Functionality. We start by formulating a DNN's functionality. In general, a DNN $f_\theta : \mathcal{X} \rightarrow \mathcal{Y}$ can be viewed as a parameterized function mapping an input $x \in \mathcal{X}$ to an output $y \in \mathcal{Y}$. The DNN structure (denoted by f) is a non-linear function and the weights θ are learned from training data, implicitly representing the rules for mapping $\mathcal{X}$ to $\mathcal{Y}$.

Well-Trained vs. Randomly-Initialized Weights. A group of well-trained weights θ should encode a fixed mapping (implicitly) defined by the training data. This encoding is gradually



(a) DNN w/ random θ (b) Input x (c) DNN w/ trained θ (d) Alternative θ

Fig. 3. Comparing DNNs with trained/random weights θ.

formed through training, during which the randomness (i.e., entropy) in weights gradually reduces [46], [22], [62], [6]. Once trained, the resulting mapping typically "focuses" more on some input elements than others (i.e., they tend to develop preferences). This distinguishes a well-trained DNN from a DNN with randomly initialized weights: the latter usually spreads its focus more evenly onto all input elements.

We illustrate this with an intuitive example in Fig. 3, where we show the computation of the same convolutional DNN with different types of weights. The randomly initialized weights before training are shown in Fig. 3(a). After training, they are updated to specialize in recognizing certain features, as shown in Fig. 3(c). When given the same input x illustrated in Fig. 3(b), the trained weights "focus" only on $x_{0,1}$, $x_{0,2}$, $x_{1,1}$, and $x_{1,2}$ because only $\theta_{0,1}$ is non-zero, while randomly initialized weights treat all input elements $x_{0,0}$–$x_{2,2}$ more equally.

Algorithm 1: Execution of a Sample Conv DNN.

```
1 function Conv2D_DNN(x, θ):
2     $kernel \leftarrow [2,2]$; $stride \leftarrow 1$; $out \leftarrow 0$;
      // slide the kernel θ over input
3     for $i \leftarrow 0$ to $3 - kernel[0]$ by $stride$ do
4         for $j \leftarrow 0$ to $3 - kernel[1]$ by $stride$ do
              // multiply θ with overlapped input
                 elements
5             for $k \leftarrow 0$ to $kernel[0] - 1$ by 1 do
6                 for $l \leftarrow 0$ to $kernel[1] - 1$ by 1 do
7                     $out \leftarrow out + x_{i+k,j+l} * \theta_{k,l}$;
8     $out \leftarrow ReLU(out)$;
9     return $out > 0$;
```

Different types of weights can cause the same DNN model to have different vulnerable bits after being compiled into executables. Consider the (simplified) execution process of the DNN model f in Fig. 3, shown in Alg. 1. Suppose the attacker flips a bit at line 6 so that the loop at line 6 always exits after the first iteration.[4] Since this keeps l fixed at 0 during the computation at line 7, it changes the prediction output (the value of out) for the trained DNN (Fig. 3(c)) from 1 to 0. But if the same flip is applied to f with randomly initialized weights (e.g., Fig. 3(a)), the prediction output remains unchanged.

Here, we are not stating that *all* possible trained weights will share this same vulnerable bit. Rather, it is more likely to find common vulnerabilities between trained weights (which have developed preferences and have lower entropy [46], [22], [62], [6]) than between trained and randomly initialized weights. Fig. 3(d) shows an example of another set of trained weights that, even though specialized to recognize different features than Fig. 3(c), also have the same vulnerable bit at line 6; in other words, this specific vulnerable bit is transferable (i.e., is a superbit) among the two sets of weights.

It is these transferable vulnerable bits, or superbits, that are the most useful to our rather constrained attacker: they allow her to launch BFA with zero knowledge of victim model weights. Specifically, if we have identified some superbits shared by multiple executables with (different) trained weights, we have high confidence that these bits are also vulnerable in the victim executable[5]; we empirically show this in Sec. VI-E. Thus, we require the set of local DNN executables $\mathcal{E}$ used for offline bit searching to consist of only executables with trained weights, not randomly initialized weights.

Constructing Fake Datasets. Recall that our threat model prohibits the attacker from accessing the victim's weights or training data; the attacker thus needs to decide the dataset(s) to use to train the models in $\mathcal{E}$. Possibly, she may arbitrarily choose a publicly available dataset, but this may hurt the transferability of the vulnerable bits found, as all local models may learn similar mappings. Or she can choose a range of public datasets to train the weights, but it remains unclear according to what metrics they should be chosen, and similar semantics (e.g., overlapping classes) between datasets may also amplify the vulnerable bits' transferability, leading to overestimated attack surface. Thus, we need a new method to obtain enough distinct trained weights that are not biased towards specific datasets involved in training the local models.

One unique opportunity, as observed in our study, is that the mapping encoded in a DNN does not have to be semantically meaningful (e.g., corresponds to real-world objects). Since we only focus on the distinction between mappings, it is unnecessary to train weights on different *real datasets*. Also, we are inspired by the observation in the machine learning community that pre-training DNNs with random noise can speed up the fine-tuning on real and meaningful datasets, because DNN weights have been "regulated" to become similar to those trained on real datasets during pre-training [63], [56]. Thus, we construct distinct "fake datasets" using random noise with different random seeds. To do so, we first randomly generate random noise as inputs and assign labels for them. Once this is done, the inputs and their labels are fixed; a fake dataset is therefore constructed. Then, when trained on a fake dataset, randomly initialized weights are gradually updated to encode the mapping in the fake dataset, forming the DNN's preferences. Since the mappings in different fake datasets are completely different, training on multiple fake datasets allows us to obtain DNN weights with distinct preferences without depending on external datasets.

V. Study Setup

DL Compilers. This research uses two DL compilers, TVM (version 0.9.0) and Glow (revision `b91adff10`), developed by Amazon and Meta (Facebook), respectively. To the best

[4] We show various ways BFA can affect program execution in Sec. VI-F.

[5] We reasonably assume the victim model also has trained weights.

TABLE II
STATISTICS OF DNN MODELS AND THEIR COMPILED EXECUTABLES EVALUATED IN OUR STUDY.

Model	#Parameters	Avg. %Acc.	Compiled Binaries	
			File Size	`.text` Size
ResNet50 [29]	23.5M	91.34	0.3-90.7M	80.8-215.7K
GoogLeNet [78]	5.5M	89.68	6.0-21.4M	221.5-337.9K
DenseNet121 [65]	7.0M	87.53	8.9-27.3M	427.3-844.5K
LeNet [45]	3.2K	98.50	78.0-90.0K	17.7-25.0K
DCGAN [67]	3.6M	-	13.7M	42.4K

TABLE III
VULNERABLE BITS IN OUR EVALUATED DNN EXECUTABLES. ℚ DENOTES QUANTIZED MODELS.

Model	Dataset	Compiler	#Bits	#Vuln.	%Vuln.	%"0→1"
ResNet50	CIFAR10	TVM	311808	8091	2.59	63.92
ResNet50	MNIST	TVM	311808	9803	3.14	61.96
ResNet50	Fashion	TVM	311808	9585	3.07	58.74
GoogLeNet	CIFAR10	TVM	903408	23136	2.56	66.49
GoogLeNet	MNIST	TVM	903408	22665	2.51	64.94
GoogLeNet	Fashion	TVM	903408	23375	2.59	62.01
DenseNet121	CIFAR10	TVM	1317424	35109	2.66	62.51
DenseNet121	MNIST	TVM	1317424	27705	2.10	70.03
DenseNet121	Fashion	TVM	1317424	30205	2.29	68.32
ℚResNet50	CIFAR10	TVM	728712	15846	2.17	55.05
ℚGoogLeNet	CIFAR10	TVM	1384904	11588	0.84	54.75
ℚDenseNet121	CIFAR10	TVM	2666280	13944	0.52	57.73
LeNet	MNIST	TVM	68800	1733	2.52	60.70
DCGAN	MNIST	TVM	220560	16634	7.54	64.64
ResNet50	CIFAR10	Glow	414600	10296	2.48	66.24
ResNet50	MNIST	Glow	414600	9614	2.32	66.35

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) %"0→1" indicates the percentage of flipping from bit 0 to 1.

of our knowledge, they represent the best DL compilers with a broad application scope and support for various hardware platforms. Both DL compilers are studied in the standard setting without any modifications. Sec. II-A has clarified the high-level workflow of DL compilation.

DNN Models and Datasets. Overall, we use five representative image classification and generative DNN models for the study. We pick ResNet50, GoogLeNet, DenseNet121, and LeNet, four popular image classification models, all of which are widely-used with varying model structures and a diverse set of DNN operators. Each of them has up to 121 layers with up to 23.5M weights. In addition, we also include the quantized versions of these models to evaluate whether their robustness against traditional weights-based BFAs still holds as DNN executables. However, we do not compile quantized models with Glow, because Glow has no support for them. As for generative models, we focus on generative adversarial networks (GANs) as they are the most popular ones. We select DCGAN, which is the backbone of nearly all modern GANs.

For the image classification models, we train them on three popular and representative datasets, CIFAR10, MNIST, and Fashion-MNIST. For DCGAN, we train it on the MNIST dataset and evaluate its outputs in aspects of image quality and semantics. These trained models, after being compiled as executables, are treated as the victim for attacks. With different compilers and configurations, we have 16 victim DNN executables as listed in Table III. To acquire trained DNNs for offline bit searching (Sec. IV-C), we also train each DNN on ten fake datasets.

Table II lists all seed DNN models, their numbers of weights and accuracies, as well as the sizes of their compiled executables and the `.text` sections. We report that the (victim) classification models have average accuracies of 91.34%, 89.68%, 87.53%, and 98.50%, respectively, for all datasets. In terms of file size, Glow-compiled executables are much smaller than TVM-compiled ones as Glow does not embed the weights into the executable but instead stores them in a separate file.

VI. EVALUATION

In this section, we report the evaluation results; we first give an overview of our key findings.

① **Pervasive Vulnerabilities in DNN Executables.** All DNN executables evaluated are pervasively vulnerable to BFAs. With reverse engineering and extensive manual efforts, we also present the characteristics of the vulnerable bits in Sec. VI-F, illustrating that vulnerable bits can originate from various binary code patterns that commonly exist in DNN executables.

② **Effective & Stealthy Corruption.** DNN executables are easily corrupted by BFAs flipping *a single bit*. This greatly enhances the *stealthiness* of our attack while also reducing its cost, making it more practical for real-world adversaries. In contrast, prior works often need to mutate a dozen bits (16.73 on average) to achieve the same attack goal.

③ **Versatile Exploitation.** We also show that DNN executables can be exploited with various end goals. In addition to contemporary works that downgrade DNN classifier accuracy via BFA, we also demonstrate attacking generative models via BFA (see evaluations in Sec. VI-C). This reveals unseen, severe consequences of our attack, such as poisoning downstream models in critical sectors trained on generated images.

④ **Transferable "Superbits."** As mentioned in Sec. IV, our (relatively weak) attacker locally constructs and profiles a set of DNN executables to identify superbits in them. We find that these bits are often (∼70%; see Sec. VI-E) transferable to the victim executable. Our observation calls for low-level, binary-centric security analysis and mitigation against BFAs, which today's techniques (mainly focusing on model weights and deployed in DL frameworks [85], [91]) are yet to cover.

⑤ **Practical Exploitation.** Our attack is successfully demonstrated on mainstream DDR4 DRAMs over all studied victim executables, whereas previous DNN-oriented BFAs only show their attack on DDR3 DRAMs.

We elaborate on our evaluation results below.

A. ① Pervasive Vulnerabilities

DNN executables compiled by mainstream DL compilers are extensively vulnerable under BFAs. Table III shows the list of binaries we have trained, compiled, and evaluated based on our study setup in Sec. V.

Overall, for both classifiers and generative models, they have from about 0.52% to 7.54% (weighted average 2.00%) of

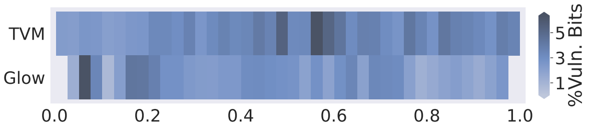


Fig. 4. Distribution of vulnerable bits in DNN executables.

their `.text` region bits vulnerable to BFA where flipping any one of the bits leads to a successful attack. Here, we consider a bit vulnerable if flipping it causes an image classification model to degrade to a random guesser (i.e., the accuracy drops to $\frac{1}{\#\text{classes}}$). For a GAN model, if, after flipping a bit, 85% of its outputs' labels changed or either the Fréchet Inception Distance (FID) [33] or average Learned Perceptual Image Patch Similarity (LPIPS) [90] score becomes higher than their 85th percentile values, we consider the bit vulnerable.

For quantized models, surprisingly, we did not find them having significantly smaller attack surfaces than their full-precision counterparts, in contrast to what prior works have suggested [88], [69]. Although they have lower percentages of vulnerable bits, their `.text` regions are significantly larger due to their more complicated structures (e.g., quantization and dequantization layers at the beginning and end of the model, and requantization layers between certain operators). As a result, the actual number of vulnerable bits is still substantial (over 10,000), leaving plenty of room for attacks.

Additionally, we report that the vulnerable bits are distributed throughout the entire `.text` region of a binary rather than being concentrated in a small region. The distribution of vulnerable bits in the `.text` region is plotted in Fig. 4, where the darkness of the color indicates the portion of vulnerable bits found in the corresponding address range inside `.text`. For Glow, the regions near the beginning and end of the `.text` region are mainly auxiliary code not involved in model inference, so we leave these regions out from evaluation. Other than that, for both TVM and Glow, vulnerable bits are distributed relatively evenly inside `.text`.

From an operator/layer viewpoint, we also show in Table VI the top 10 operators containing the vulnerable bits in the ResNet50 CIFAR10 executable compiled by TVM ("Default Executable" columns), where no single operator dominates the existence of vulnerable bits. As TVM fuses neighboring operators wherever possible (indicated by the "+" in the table), we further build and analyze the same executable with fusion disabled ("With Fusion Disabled" columns); this executable confirms our observation as well. In general, these widely spread out vulnerable bits translates to higher success rates for attackers, as the "one bit per page" constraint (see Sec. IV-B) will have much less impact on them.

In Table IV, we explore the attack surface of BFA on different executable variants compiled by TVM (which are unsupported by Glow). We compare the default executable with two other variants: (1) No Fusion, which turns off the fusion and other optimizations in TVM, and (2) No AVX2, which turns off AVX2 instructions in the executable. Overall, we find that different variants may have different percentages of vulnerable bits, but this number is at approximate the same level (2-4%) for all cases. Since the TVM optimization pipeline simplifies and fuses many operators in the computational graph [15], the large amount of unfused operators in an unoptimized executable may introduce more structures vulnerable to BFAs, such as loops and exploitable loop variables, hence more vulnerable bits. On the other hand, turning off AVX2 replaces all AVX2 instructions in an executable with simpler SSE instructions. While there will be more instructions as SSE is less vectorized, the new instructions are shorter, shrinking the binaries' `.text` regions. Also, we noticed that the opcode bits in these SSE instructions are more "flippable" than those in AVX2 instructions, i.e., when flipped, an opcode is more likely to become another *valid* opcode that is still valid, as can be seen by comparing the most common flip types before and after turning off AVX2 in Table V. This is likely because shorter instructions have a smaller opcode space, so different opcodes are closer to each other (in terms of Hamming distance).

TABLE IV
VULNERABILITY STATISTICS OF DIFFERENT EXECUTABLE VARIANTS COMPILED BY TVM.

Model	Dataset	Variant	#Bits	#Vuln	%Vuln	%"0→1"
ResNet50	CIFAR10	Default	311808	8091	2.59	63.92
ResNet50	CIFAR10	No Fusion	809744	27328	3.37	58.78
ResNet50	CIFAR10	No AVX2	268992	10999	4.09	64.27
ResNet50	MNIST	Default	311808	9803	3.14	61.96
ResNet50	MNIST	No Fusion	809744	26942	3.33	55.94
ResNet50	MNIST	No AVX2	268992	11832	4.40	64.24

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) %"0→1" indicates the percentage of flipping from bit 0 to 1.

TABLE V
TOP 10 MOST COMMON FLIP TYPES IN AVX2 AND NON-AVX2 BINARIES. "PCT." STANDS FOR PERCENTAGE.

Rank	AVX2 Binaries		Non-AVX2 Binaries	
	Instruction (Flip Type)	Pct. (%)	Instruction (Flip Type)	Pct. (%)
1	VMOVUPS (Data)	12.34	MULPS (Opcode)	16.96
2	MOV (Data)	8.82	MULPS (Data)	15.96
3	ADD (Data)	6.23	ADDPS (Opcode)	9.12
4	VBROADCASTSS (Data)	5.83	MOV (Data)	8.02
5	VXORPS (Data)	5.17	MOVAPS (Opcode)	5.63
6	VFMADD231PS (Data)	4.88	XORPS (Data)	4.88
7	LEA (Data)	4.86	ADD (Data)	4.37
8	VMOVAPS (Data)	3.47	ADDPS (Data)	4.07
9	VADDPS (Opcode)	3.35	MOVAPS (Data)	3.96
10	VADDPS (Data)	3.31	MOVSS (Opcode)	3.78

B. ② *Effective & Stealthy Corruption*

As mentioned in the "conditions for vulnerable bits" in Sec. IV-B, all of our findings are vulnerable bits that can be used to deplete full DNN model intelligence with only a single-bit BFA. For example, flipping bit 1 of the byte at offset `0x1022f6` in the first binary in Table III causes the model's prediction accuracy to drop from 87.20% to 11.00%, equivalent to a random guesser. To the best of our knowledge, this is the first work to report such single-bit corruptions in DNN models. We believe the ability to corrupt a model using one single-bit flip is an important motivation for real-

TABLE VI

TOP 10 OPERATORS CONTAINING VULNERABLE BITS IN THE TVM RESNET50 CIFAR10 EXECUTABLE. "PCT." STANDS FOR PERCENTAGE.

Rank	Default Executable		With Fusion Disabled	
	Operator	Pct. (%)	Operator	Pct. (%)
1	Conv2d + Add + ReLU 0	13.32	Conv2d 2	7.30
2	Conv2d + Add + ReLU 2	10.37	Conv2d 0	6.13
3	Conv2d + Add + Add + ReLU 0	5.96	Conv2d 3	5.51
4	Conv2d + Add + Add + ReLU 2	5.55	Conv2d 10	5.25
5	Conv2d + Add 0	5.39	Conv2d 7	4.90
6	Conv2d + Add + Add + ReLU 1	4.52	Conv2d 13	4.56
7	Conv2d + Add + ReLU 3	4.24	Conv2d 6	3.58
8	Adaptive-avgpool 0	3.78	Conv2d 15	3.28
9	Conv2d + Add + ReLU 10	3.65	Conv2d 4	3.21
10	Conv2d + Add + Add + ReLU 3	3.32	Conv2d 5	3.12

TABLE VII

NUMBER OF VULNERABLE BITS BY OUTPUT CLASS. THE LAST TWO ROWS ARE GLOW-COMPILED EXECUTABLES WHEREAS THE REST ARE TVM-COMPILED.

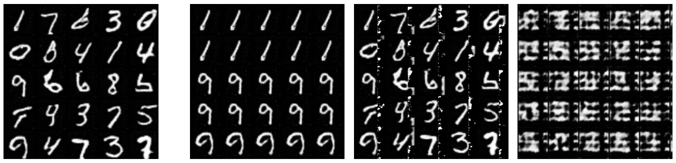
Model	Dataset	#Vulnerable Bits by Output Class									
		0	1	2	3	4	5	6	7	8	9
ResNet50	CIFAR10	742	447	3192	881	121	1509	381	67	249	502
ResNet50	MNIST	12	7699	33	17	8	77	14	150	284	1509
ResNet50	Fashion	0	8543	96	6	2	87	71	19	336	425
GoogLeNet	CIFAR10	1747	2022	8430	1538	614	1814	1088	918	1290	3675
GoogLeNet	MNIST	188	1600	1226	1730	256	2991	2133	526	2062	9953
GoogLeNet	Fashion	6165	388	722	302	2989	1123	505	1366	417	9398
DenseNet121	CIFAR10	604	23425	1681	1881	2328	504	1321	234	1913	1218
DenseNet121	MNIST	3563	895	837	4506	6192	335	2027	233	3740	5377
DenseNet121	Fashion	16718	1014	551	1282	152	1945	5402	26	2114	1001
ℚResNet50	CIFAR10	866	1534	4375	2409	397	3127	1295	488	389	966
ℚGoogLeNet	CIFAR10	3612	165	1707	2336	266	348	394	600	931	1229
ℚDenseNet121	CIFAR10	1546	3295	1051	2794	1840	74	1021	578	705	1040
ResNet50	CIFAR10	1369	1023	1940	1278	331	2225	496	357	647	630
ResNet50	MNIST	36	7010	47	39	31	102	23	130	305	1891

world attackers because, under the assumption that BFAs are mostly instantiated using RH, which is a probabilistic process, it greatly reduces the cost for the attack and increases the success rate, as we will see in our practical attack experiments in Sec. IV-B. It also largely reduces the risk of being detected by the victim (i.e., more stealthy), as the corruption is much more subtle than the case of multi-bit corruption.

C. ③ *Versatile End-Goals*

As mentioned earlier, the consequences BFA can cause are not limited to downgrading a classification model's accuracy. We observe that BFA also shows a high potential to *manipulate* DNN executables' prediction results or generative outputs, and this phenomenon also extensively exists in our study.

First, in terms of classification models, Table VII shows a summary of different predicted classes that can be controlled by single-bit BFA. When a model is pinned to a class by BFA, it has the highest probability of outputting that class for any input, granting attackers the ability to control model outputs. We notice that, for most of the models in the list, there are frequently hundreds or even thousands of vulnerable bits that can pin the model to each of the classes, although in rare cases, there will be few or no bits available for a specific class. While this may not be a targeted BFA (T-BFA) in the standard sense [14] due to its non-deterministic nature, we should point out that no existing work has demonstrated practical T-BFA that can pin a model's output using one bit, whereas all of our findings are achieved via single-bit BFA. Thus, we see this as a "high risk, high return" strategy for sophisticated attackers.



(a) Before BFA (b) Three different types of outcomes after BFA

Fig. 5. Output samples of DCGAN before and after BFA.

For GAN, Fig. 5(a) shows the original output of our DCGAN model before being corrupted by BFA, and Fig. 5(b) shows three different types of outcomes the model produces when given the same input after flipping three different bits in the model. Among the three types, the first one may be the most interesting: not only does it almost completely change the semantics of the output, but it also pins the model output to only two semantic classes (1 and 9). Also notice the two styles of the 9's in the output sample: they suggest that this flip is not simply causing duplications in the output, but is manipulating the model's learned semantics as well. Then, the second type degrades the output image quality while preserving the semantics, and the third destroys both the semantics and image quality. If the attacked GAN is used for augmenting a DNN model, in the first case, it is anticipated that the augmented DNN will tend to predict "1" or "9" for any input since its training data are dominated by 1's and 9's. In the other two cases, the augmented DNN should also be largely downgraded because its training data are less recognizable.

Out results review a new attack angle and its severe consequence: generative models are often adopted for data augmentation (e.g., for medical image analysis [79], [28], [23], [11], [55]), and manipulating the generated images can introduce bias into the augmented datasets, making the augmented DNN biased. For example, by leveraging BFA, we can force a chest X-ray image generator to always generate benign X-ray images. Then, the DNN augmented using this manipulated dataset will tend to predict most inputs as "benign."

D. ④ *Superbits*

We find some vulnerable bits transferable among different DNN executable even though they are trained on different datasets but just sharing the same DNN structure; we call these bits *superbits*. In Table VIII, we summarize the existence of superbits over different models: for each model structure, we train and compile three executables, each on a different dataset (CIFAR10, MNIST, or Fashion-MNIST); after obtaining the executables, we search for superbits across all 3 executables sharing the same DNN structure. Generally, comparing with the results in Table III, we find that about half of the vulnerable bits found in one DNN executable trained on one dataset also exist in the executables trained on the other two datasets.

Since a superbit is located at the same offset and has the same flip direction in all executables that share it, an attacker will find it much more convenient to launch BFAs if she can find superbits shared by the victim DNN executable. In

TABLE VIII
STATISTICS OF SUPERBITS IN DNN EXECUTABLES OF THE SAME STRUCTURE BUT DIFFERENT WEIGHTS.

Structure	Datasets & Weights	Compiler	#Superbits	%Superbits
ResNet50	CIFAR10 / MNIST / Fashion	TVM	4334	1.61
GoogLeNet	CIFAR10 / MNIST / Fashion	TVM	12422	1.38
DenseNet121	CIFAR10 / MNIST / Fashion	TVM	18349	1.39
$\mathbb{Q}$ResNet50	CIFAR10 / MNIST / Fashion	TVM	7579	1.04
$\mathbb{Q}$GoogLeNet	CIFAR10 / MNIST / Fashion	TVM	1994	0.14
$\mathbb{Q}$DenseNet121	CIFAR10 / MNIST / Fashion	TVM	6517	0.24
ResNet50	CIFAR10 / MNIST / Fashion	Glow	5223	1.26

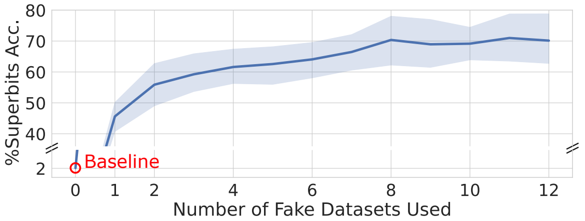


Fig. 6. Relation between the number of fake datasets (Sec. IV-C) used and the accuracy of the superbits found, shown as the average for *all* attacked executables with its 95% confidence interval. The baseline case of not using fake datasets ("0") is also included.

fact, we have shown in Sec. IV-B that it is indeed feasible to find superbits that are highly likely to be shared by a set of DNN executables the attacker possesses *plus* the victim DNN executable, and we described a systematic search method to achieve this effectively and efficiently. In Sec. VI-E, we further use the existence of superbits and our novel search method to launch practical BFAs without relying on knowledge about the victim model's weights. Finally, in our case study in Sec. VI-F, we will provide more insights into why they can effectively disrupt the behavior even of different DNN executables.

E. ⑤ Practical Attack

This section demonstrates that practical BFAs can be launched against DNN executables. We follow the steps in Blacksmith [40] to assess the practical exploitations, and run our experiments on a server with an Intel i7-8700 CPU and a Samsung 8GB DDR4 DRAM module without any hardware modifications. Before launching our attacks, we extend Blacksmith to launch our RH attacks on DDR4 more smoothly: we find Blacksmith's timing function rather easily affected by noise in our preliminary experiments, making it hard to distinguish between DRAM accesses with and without row conflicts. We thus replace the timing function with the one in TRRespass, which we find to be more resilient to noise on our platform. We also replace Blacksmith's Hammertime framework [80] to adapt to our work. Our practical attack experiment covers in total 9 different DNN executables to evaluate the attack effectiveness on different model structures, datasets, and compilers. For each executable, we launch the attack five times and collect the results.

We first perform memory templating (Sec. IV-B) by running Blacksmith with default settings to obtain memory templates in a 256MB memory region. The sweep identified a total of 17,366 flippable bits in the region, 8,855 of which are 0→1 flips. We then obtain the superbits $\mathcal{S}_{\mathcal{E}}$ using the search method in Sec. IV; these are the bits we will attempt to flip during our attacks. We report that the local profiling stage takes on average 41.4 hours per executable (not adapted for parallelism). Note that unlike prior BFAs that re-do the profiling for every victim DNN, our profiling is a *one-time* effort and applies to all same-structure-different-weights DNNs.

TABLE IX
STATISTICS OF THE 5 ATTACK RUNS ON 9 EXECUTABLES. THE LAST ROW IS FOR GLOW-COMPILED EXECUTABLE WHEREAS THE REST ARE FOR TVM-COMPILED.

Model	Dataset	#Flips	#Crashes	%Acc. Change
ResNet50	CIFAR10	1.4	0.0	87.20 → 10.00
GoogLeNet	CIFAR10	1.4	0.0	84.80 → 10.00
DenseNet121	CIFAR10	1.0	0.0	80.00 → 11.40
DenseNet121	MNIST	1.2	0.0	99.10 → 11.20
DenseNet121	Fashion	1.2	0.0	92.50 → 10.60
$\mathbb{Q}$ResNet50	CIFAR10	1.6	0.0	86.90 → 9.60
$\mathbb{Q}$GoogLeNet	CIFAR10	1.4	0.0	84.60 → 11.20
$\mathbb{Q}$DenseNet121	CIFAR10	1.6	0.0	78.50 → 10.20
ResNet50	CIFAR10	1.4	0.0	78.80 → 10.00

To determine the number of "fake datasets" used to calculate $\mathcal{S}_{\mathcal{E}}$ (Sec. IV-C), we plot in Fig. 6 the relationship between the number of fake datasets used and the accuracy of the superbits found, i.e., how many bits in $\mathcal{S}_{\mathcal{E}}$ are actually also vulnerable in the victim executable. For comparison, we also show the baseline case where the attacker does not use fake datasets, but simply selects random bits as superbits to use in the attack. We observed that, the attacker is able to confidently find superbits when using 8 or more fake datasets (with ∼70% accuracy), and the confidence interval is the tightest at 10 datasets. Randomly selected bits have significantly lower probabilities (∼2%) of being transferable to the victim executable. We thus use 10 fake datasets to obtain $\mathcal{S}_{\mathcal{E}}$.

The statistics of the attack results are shown in Table IX. On average, we successfully degrade each victim executable to a random guesser (prediction accuracy of 10%) with 1.4 flip attempts while causing no crashes at all, regardless of the original prediction accuracy of the victim executable. In the case of DenseNet121 on CIFAR10, we consistently succeed with just one flip attempt in all five runs, decreasing its accuracy from 80.00% to 11.40%, ruining its inference capabilities. Somewhat surprising are the results for quantized models. Recall that, quantized models are considered substantially harder to attack with BFA, requiring 2× to 23× more flips for complete intelligence depletion [88]. We however find that, after being compiled into DNN executables, quantized models require just 1.4 to 1.6 flips to be successfully attacked, which is only slightly higher than the figures for full-precision DNN executables. The worst case was found in one run for the quantized DenseNet121 model, where four flips were required before successfully achieving the goal. We compare our results with existing work in Sec. VII-A. Our observation suggests that BFA is a severe *and* practical threat to DNN executables.

F. Case Study

To understand the root causes behind a single-bit flip compromising the complete intelligence of a DNN executable as well as provide inspiration for future countermeasures (more

TABLE X
CLASSIFICATION OF MANUALLY ANALYZED BFA CASES.

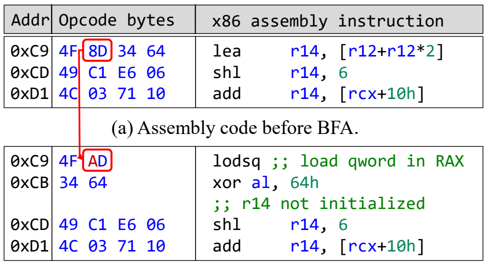
Bit Type	Total	Data Flow	Control Flow	Data Align	Inst Align
Non-Superbit	30	13	14	2	1
Superbit	30	5	8	12	1

discussion in Sec. VII-B), we randomly analyze 60 cases, 30 each for non-superbits and superbits, from our previous results. After an extensive manual study, we list in Table X four categories of causes for single-bit corruption, including

- Broken Data Flow: the calculation of a specific layer's input or output address is corrupted, causing the inputs (outputs) to be read from (written to) wrong memory regions.
- Broken Control Flow: a condition is changed to be always false, causing the corresponding calculation branch to be skipped, producing a large number of incorrect outputs.
- Broken Data Alignment: the offset of memory read/write instructions is deviated, causing data to be read or written in an unaligned manner.
- Broken Instruction Alignment: the bit flip converts bytes in the .text section for alignment purposes into instructions, causing subsequent instructions to be corrupted.

Our case study reveals common code patterns across models and datasets. The study also shows why existing defenses (Sec. VII-A) are not effective: they focus on the protection of victim model weights and are unable to detect attacks like ours, which target program parts *other than* the weights. Although the analyzed cases come from the same DNN executable (i.e., LeNet1), in our observation, the results are applicable to other DNN executables compiled by TVM and Glow and can offer a comprehensive understanding of the causes behind successful BFA. We now discuss one representative case for each category.

Broken Data Flow. We observed many different patterns of how a single bit flip could break the data flow of model inference. Here, we provide one example related to the parallelism of DNN executables. Typically, a DNN executable runs in parallel where multiple threads are launched to perform computation for a DNN layer, each thread computing a portion of the output. When a thread is initialized, its corresponding output offset is calculated using its thread ID.



```
Addr  Opcode bytes    x86 assembly instruction
0xC9  4F 8D 34 64     lea     r14, [r12+r12*2]
0xCD  49 C1 E6 06     shl     r14, 6
0xD1  4C 03 71 10     add     r14, [rcx+10h]
```

(a) Assembly code before BFA.

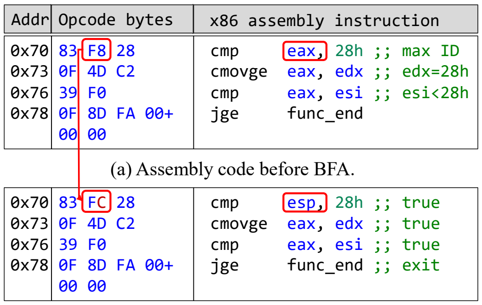
```
0xC9  4F AD           lodsq ;; load qword in RAX
0xCB  34 64           xor al, 64h
                      ;; r14 not initialized
0xCD  49 C1 E6 06     shl     r14, 6
0xD1  4C 03 71 10     add     r14, [rcx+10h]
```

(b) Assembly code after BFA.

Fig. 7. Case 1: the multi-thread data flow is broken.

Consider the example in Fig. 7, where r14 is a register storing the offset and [rcx+10h] one storing the base address of the output. Before BFA, the offset (r14) is calculated as base_address+r12*192 (r12 stores the thread ID). However, after BFA, the instruction at 0xC9 is split into two instructions irrelevant to r14 (at 0xC9 and 0xCB), resulting in all threads simultaneously writing to the same output region.

Broken Control Flow. The number of threads launched by DNN executables is determined by a predefined environment variable. When there are more threads than required by a DNN layer (e.g., a convolutional layer), redundant threads will directly jump to the function end after thread ID checking.



```
Addr  Opcode bytes    x86 assembly instruction
0x70  83 F8 28        cmp     eax, 28h ;; max ID
0x73  0F 4D C2        cmovge  eax, edx ;; edx=28h
0x76  39 F0           cmp     eax, esi ;; esi<28h
0x78  0F 8D FA 00+    jge     func_end
      00 00
```

(a) Assembly code before BFA.

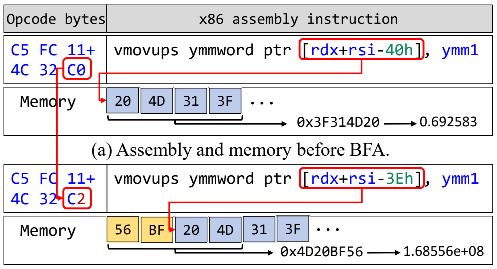
```
0x70  83 FC 28        cmp     esp, 28h ;; true
0x73  0F 4D C2        cmovge  eax, edx ;; true
0x76  39 F0           cmp     eax, esi ;; true
0x78  0F 8D FA 00+    jge     func_end ;; exit
      00 00
```

(b) Assembly code after BFA.

Fig. 8. Case 2: the control flow is broken.

This check, however, is vulnerable to BFA. As shown in Fig. 8, the instruction at 0x70 originally compares 0x28 with eax; after BFA, it compares with esp which always stores a very large stack address. Thus, the following comparisons are always true, making all threads skip the execution.

Broken Data Alignment. In the computation of DNN executables, all floating-point numbers are represented as 4-byte aligned data. However, such alignment can be easily violated with even only a single-bit flip. As shown in Fig. 9, the vmovups instruction will move 32 bytes of data from the ymm1 register to the memory. The BFA increases the offset of the target memory address by 2 bytes, resulting in unaligned data written into memory. The next time the data is read from memory in an aligned manner, the corrupted data will be interpreted as an extremely large float value (e.g., 1e8). These large numbers will propagate in the process of DNN model inference and dominate the final result.



```
Opcode bytes    x86 assembly instruction
C5 FC 11+       vmovups ymmword ptr [rdx+rsi-40h], ymm1
4C 32 C0
Memory          20 4D 31 3F ...
                  0x3F314D20 -> 0.692583
```

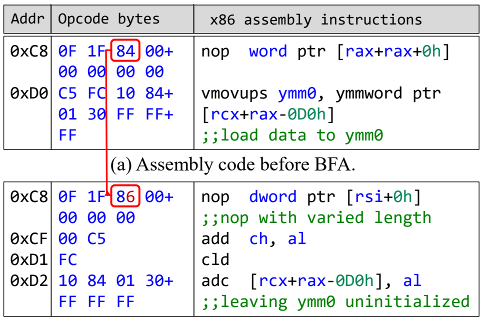
(a) Assembly and memory before BFA.

```
C5 FC 11+       vmovups ymmword ptr [rdx+rsi-3Eh], ymm1
4C 32 C2
Memory          56 BF 20 4D 31 3F ...
                  0x4D20BF56 -> 1.68556e+08
```

(b) Assembly and memory after BFA.

Fig. 9. Case 3: the data alignment is broken.

Broken Instruction Alignment. During compilation, nop instructions are often used to align instruction addresses. Similar cases are observed in DNN executables. Consider the example in Fig. 10, a nop instruction is used to align the next instruction to address 0xD0. Nevertheless, after a single-bit flip, the nop instruction is converted into a shorter variant, leaving its last byte (at 0xCF) being recognized as the start of



```
Addr | Opcode bytes | x86 assembly instructions
0xC8 | 0F 1F 84 00+ | nop  word ptr [rax+rax+0h]
     | 00 00 00 00  |
0xD0 | C5 FC 10 84+ | vmovups ymm0, ymmword ptr
     | 01 30 FF FF+ | [rcx+rax-0D0h]
     | FF           | ;;load data to ymm0
```

(a) Assembly code before BFA.

```
0xC8 | 0F 1F 86 00+ | nop  dword ptr [rsi+0h]
     | 00 00 00     | ;;nop with varied length
0xCF | 00 C5        | add  ch, al
0xD1 | FC           | cld
0xD2 | 10 84 01 30+ | adc  [rcx+rax-0D0h], al
     | FF FF FF     | ;;leaving ymm0 uninitialized
```

(b) Assembly code after BFA.

Fig. 10. Case 4: the instruction alignment is broken.

TABLE XI
A COMPARISON OF ATTACK PERFORMANCE WITH PRIOR WORKS. FOR MITIGATIONS, ○, ◐, AND ● DENOTE NO, PARTIAL, AND FULL MITIGATION, RESPECTIVELY.

Work	Attack Target	Avg. #Flips	Mitigable by				
			Q [92]	A [85]	D [13]	W [49]	N [50]
BFA [69]	Weights	14.3	◐	◐	●	●	●
T-BFA (N-to-1) [71]	Weights	23.63	◐	◐	●	●	●
DeepHammer [88]	Weights	12.25	◐	◐	●	●	●
Ours	Structure	1.4	○	○	○	○	○

an add instruction. The instruction alignment is thus broken, leading to an uninitialized register (ymm0) being used in the computation. In this case, the presence of nan values in ymm0 directly destroys subsequent DNN model inference.

VII. Discussion

A. Comparison Against Existing Attacks and Defenses

Existing Attacks. We list in the rows of Table XI the attack performance of prior state-of-the-art BFAs against DNN models running on DL frameworks (data from original papers), as well as their mitigability by existing defenses. We include BFA [69] as the standard gradient-based method for flipping weight bits, T-BFA [71] as its targeted variant, and DeepHammer [88] as its RH-specific counterpart. We find that these attacks require 12.25 to 23.63 flips on average to achieve the attack goals, while our method only needs 1.4 flips.

Existing Defenses. We also compare our attack (together with other attacks) against existing defenses in the columns of Table XI. We include representative works in both the passive and active defense categories. Passive defenses like quantization [92] and Aegis [85] transform the protected models to enhance their robustness against BFAs; they thus only partially mitigate weights-based attacks by increasing the number of required flips. The use of randomized multi-exit structures in Aegis also makes it inapplicable to DNN executables because they are unsupported by DL compilers. And as shown in Sec. VI-E, change in model precision (quantized or not) does not affect our attack performance. On the other hand, active defenses aim to detect BFAs as attackers are making attempts. DeepAttest [13] injects fingerprints into model weights through model fine-tuning and uses specialized trusted hardware to verify it during inference. Weight-encoded detection [49] and NeuroPots [50] determine important weights using gradients, encode keys into them, and extract the keys at runtime for verification. While these methods can detect existing weights-based BFAs which also use gradient information to pick weights to flip, our attack does not modify weights or rely on gradients, and thus is not detectable. In addition, porting these active defenses to DNN executables poses challenges, as their runtime verification mechanisms are compiled into DNN executables as well and may also become attackers' targets. As a result, our attack slips past all existing defenses and calls for new defense mechanisms tailored for DNN executables.

B. Future Defense Directions

Based on our findings, we discuss potential defenses on two levels. First, we anticipate the potential of using existing RH defenses [41], [32], [83], [20] to lower BFA risks specific to this paper's threat model which assumes a weak attacker with limited knowledge and constrained by current RH techniques. A classic defense is to install ECC memory modules. However, this only lowers the risks of BFA without eliminating them [18], and platforms like embedded devices that do not support ECC memory may still be vulnerable [82]. Compile-time code obfuscation can be designed and implemented to prevent attackers from producing same-structure-different-weights executables for local profiling, although it does not protect against cases where public model executables are used or where attackers have full knowledge of the victim, as assumed by existing works [88], [69], [71]. MLaaS providers may also enforce stricter security policies (e.g., restricting the eviction of vulnerable memory pages) to prevent current RH techniques from working, but performance or benefits from resource sharing may get discounted.

On a higher level, however, BFA on DNN executables should be studied independently of the underlying error injection techniques like RH, since diverse sources ranging from a depleted power supply to high-energy light beams can also trigger bit flips [7]. For traditional DNN models on DL frameworks, efforts have been made to protect them against generic fault injection attacks on weights [30], [85], [47], [50]. But as we have discussed in Sec. VII-A, they are either inapplicable to DNN executables or cannot protect against structure-based attacks on DNN executables. We thus envision that low-level, binary-centric security mechanisms are needed to substantially reduce the attack surface. Since flipping vulnerable bits mainly corrupts data and control flow (as discussed in Sec. VI-F), yet another potential defense might be implementing DNN-executable-specific data/control flow integrity checks [3], [12]. Currently, some work has been done to detect abnormal neuron activation in DNN executables at runtime by comparing them with reference values or computing gradient-based metrics [16], but it is still unclear how BFA defenses can benefit from it, and how control flow integrity checks should be designed for DNN executables. In summary, designing comprehensive BFA defense schemes for DNN executables is still an open problem, and we leave it as future work.

C. Generality and Extension

Aligned with prior BFAs [88], [35], this work mostly studies computer vision models; some other types of DNN models are not discussed. For instance, NLP models may contain recurrent structures such as RNN and long short-term memory (LSTM) [34] to handle sequential inputs. We tentatively tried to evaluate our methods on these models but found that both TVM and Glow show immature support for them. However, our method should generalize to these other models since the vulnerable bits and related binary code patterns are model/operator-agnostic (see Sec. VI-F).

While this paper mainly focuses on the x86 architecture, our provisional qualitative results show that DNN executables compiled for other instruction set architectures (ISAs) like ARM also manifest similar BFA vulnerabilities. As various architectures rapidly gain popularity [2], we plan to fully extend our study to non-x86 architectures in the future.

VIII. Related Work

To date, RH attacks have been successfully applied in a wide range of exploitations and in different directions, including Linux privilege escalation [76], browser-based remote attacks [27], mobile-platform-specific attacks [82], and others [42], [40], [18]. Gruss et al. [26] proposed the idea of hammering in the code region of specific executables to corrupt their execution logic. While we also target executable code regions, we specialize to the case of DNN executables and provide an automated searcher for vulnerable bits (details in Sec. IV). Meanwhile, the attacks have also led to the rise of corresponding security mitigation techniques [41], [32], [83], [20]. In DDR4 DRAM, the Target Row Refresh (TRR) has been widely adopted by manufacturers as an on-chip mitigation for RH attacks, but research has shown that the protection is incomplete [24], [40]. GuardION [83] is a software-based defense designed to prevent DMA-based RH attacks on ARM platforms primarily running Android OS. Copy-on-Flip [20] utilizes the error correction events generated by systems with ECC memory to detect and mitigate RH attacks, at the cost of performance overhead.

For BFAs targeting DNN models, Rakin et al. [69] proposed a progressive bit search algorithm to find the most vulnerable bits in the weights of a model: intra-layer bit searching is performed for each layer and the top weight bits with the largest gradients are recorded. The most vulnerable bits across all layers are then obtained as the model-wise top bits to flip. Building on this, Yao et al. [88] proposed DeepHammer to consider RH-specific constraints and use RH to flip the identified bits. A targeted variant of the progressive search method was put forward to achieve more advanced attack goals using RH on DNN models [71]. Rakin et al. also proposed to steal DNN model weights with RH [68]. We have covered related work on defending DNN models against BFAs in Sec. VII-A and Sec. VII-B.

IX. Conclusion

We launch the first systematic study on the attack surface of BFA on DNN executables. We show that DNN executables are pervasively vulnerable to BFAs, and can be exploited in a highly practical manner. Our findings call for security mechanisms in future DL compilation toolchains.

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. The authors would also like to thank Patrick Jattke for providing data and suggestions on the use of Blacksmith.

References

[1] Research Artifact. https://sites.google.com/view/exe-single-bit-bfa.

[2] Arm's growing cloud server momentum. https://www.forbes.com/sites/stevemcdowell/2023/02/26/arms-growing-cloud-server-momentum, 2023.

[3] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

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

[5] 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.

[6] Pierre Baldi and Peter J Sadowski. Understanding dropout. *Advances in neural information processing systems*, 26, 2013.

[7] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[8] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. Csi nn: reverse engineering of neural network architectures through electromagnetic side channel. In *Proceedings of the 28th USENIX Conference on Security Symposium*, pages 515–532, 2019.

[9] 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*, CCS '18, pages 2204–2206. Association for Computing Machinery.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[11] Francesco Calimeri, Aldo Marzullo, Claudio Stamile, and Giorgio Terracina. Biomedical data augmentation using generative adversarial neural networks. In *Artificial Neural Networks and Machine Learning–ICANN 2017: 26th International Conference on Artificial Neural Networks, Alghero, Italy, September 11-14, 2017, Proceedings, Part II 26*, pages 626–634. Springer, 2017.

[12] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[13] 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.

[14] 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.

[15] 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.

[16] Yanzuo Chen, Yuanyuan Yuan, and Shuai Wang. OBSAN: An Out-Of-Bound Sanitizer to Harden DNN Executables. In *NDSS 2023*.

[17] 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.

[18] 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.

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

[20] Andrea Di Dio, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks. In *Proceedings 2023 Network and Distributed System Security Symposium*. Internet Society.

[21] Robert Elder. What causes bit flips in computer memory? https://blog.robertelder.org/causes-of-bit-flips-in-computer-memory/#row-hammer, 2023.

[22] Gianni Franchi, Andrei Bursuc, Emanuel Aldea, Séverine Dubuisson, and Isabelle Bloch. Tradi: Tracking deep neural network weight distributions. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVII 16*, pages 105–121. Springer, 2020.

[23] Maayan Frid-Adar, Idit Diamant, Eyal Klang, Michal Amitai, Jacob Goldberger, and Hayit Greenspan. Gan-based synthetic medical image augmentation for increased cnn performance in liver lesion classification. *Neurocomputing*, 321:321–331, 2018.

[24] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Trrespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*.

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

[26] 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.

[27] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*, pages 300–321. Springer, 2016.

[28] Changhee Han, Hideaki Hayashi, Leonardo Rundo, Ryosuke Araki, Wataru Shimoda, Shinichi Muramatsu, Yujiro Furukawa, Giancarlo Mauri, and Hideki Nakayama. Gan-based synthetic brain mr image generation. In *2018 IEEE 15th international symposium on biomedical imaging (ISBI 2018)*, pages 734–738. IEEE, 2018.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[30] 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.

[31] Kevin Hector, Pierre-Alain Moëllic, Jean-Max Dutertre, and Mathieu Dumont. Fault injection and safe-error attack for extraction of embedded neural network models. In *European Symposium on Research in Computer Security*, pages 644–664. Springer, 2023.

[32] Nishad Herath and Anders Fogh. These are not your grand daddys cpu performance counters–cpu hardware performance counters for security. *Black Hat Briefings*, 2015.

[33] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[34] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[35] 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.

[36] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *ASPLOS*, pages 385–399, 2020.

[37] 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.

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

[39] 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.

[40] 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.

[41] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural support for mitigating row hammering in dram memories. *IEEE Computer Architecture Letters*, 14(1):9–12, 2014.

[42] 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.

[43] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711. IEEE, 2020.

[44] 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.

[45] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[46] Jinsol Lee and Ghassan AlRegib. Gradients as a measure of uncertainty in neural networks. In *2020 IEEE International Conference on Image Processing (ICIP)*, pages 2416–2420. IEEE, 2020.

[47] 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.

[48] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing rowhammer faults through network requests. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 710–719. IEEE, 2020.

[49] 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.

[50] 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.

[51] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on cpus. In *USENIX ATC*, pages 1025–1040, 2019.

[52] Zhibo Liu, Yuanyuan Yuan, Yanzuo Chen, Sihang Hu, Tianxiang Li, and Shuai Wang. Deepcache: Revisiting cache side-channel attacks in deep neural networks executables. In *CCS 2024*.

[53] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, Xiaofei Xie, and Lei Ma. Decompiling x86 Deep Neural Network Executables. In *USENIX Security 2023*.

[54] 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.

[55] Ali Madani, Mehdi Moradi, Alexandros Karargyris, and Tanveer Syeda-Mahmood. Chest x-ray generation and data augmentation for cardiovascular abnormality classification. In *Medical imaging 2018: Image processing*, volume 10574, pages 415–420. SPIE, 2018.

[56] Hartmut Maennel, Ibrahim M Alabdulmohsin, Ilya O Tolstikhin, Robert Baldock, Olivier Bousquet, Sylvain Gelly, and Daniel Keysers. What do neural networks learn when trained with random labels? *Advances in Neural Information Processing Systems*, 33:19693–19704, 2020.

[57] Timothy Prickett Morgan. INSIDE FACEBOOK'S FUTURE RACK AND MICROSERVER IRON. https://www.nextplatform.com/2020/05/14/inside-facebooks-future-rack-and-microserver-iron/, 2020.

[58] Nvidia. NVVM IR. https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html, 2021.

[59] 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.

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

[61] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *ACM Asia CCS*, pages 506–519, 2017.

[62] Georg Pichler, Pierre Jean A Colombo, Malik Boudiaf, Günther Koliander, and Pablo Piantanida. A differential entropy estimator for training neural networks. In *International Conference on Machine Learning*, pages 17691–17715. PMLR, 2022.

[63] Vinaychandran Pondenkandath, Michele Alberti, Sammer Puran, Rolf Ingold, and Marcus Liwicki. Leveraging random label memorization for unsupervised pre-training. *Workshop of Integration of Deep Learning Theories at Conference on Neural Information Processing Systems (NIPS)*, 2018.

[64] Ivan Puddu, Moritz Schneider, Daniele Lain, Stefano Boschetto, and Srdjan Čapkun. On (the lack of) code confidentiality in trusted execution environments. *arXiv preprint arXiv:2212.07899*, 2022.

[65] Pytorch. Dense Convolutional Network (DenseNet). https://pytorch.org/hub/pytorch_vision_densenet/, 2021.

[66] 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.

[67] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[68] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1157–1174. IEEE, 2022.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] Mauro Ribeiro, Katarina Grolinger, and Miriam A.M. Capretz. Mlaas: Machine learning as a service. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 896–902, 2015.

[75] 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.

[76] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15:71, 2015.

[77] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

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

[79] Youssef Skandarani, Pierre-Marc Jodoin, and Alain Lalande. Gans for medical image synthesis: An empirical study. *Journal of Imaging*, 9(3):69, 2023.

[80] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*, pages 47–66. Springer, 2018.

[81] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Sec'16*.

[82] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689, 2016.

[83] Victor Van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. Guardion: Practical mitigation of dma-based rowhammer attacks on arm. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, June 28–29, 2018, Proceedings 15*, pages 92–113. Springer, 2018.

[84] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[85] 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. *arXiv preprint arXiv:2302.13520*, 2023.

[86] 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.

[87] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn dnn architectures. In *USENIX Sec'20*.

[88] 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.

[89] Honggang Yu, Haocheng Ma, Kaichen Yang, Yiqiang Zhao, and Yier Jin. Deepem: Deep neural networks model recovery through em side-channel information leakage. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020.

[90] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 586–595, 2018.

[91] Ranyang Zhou, Sabbir Ahmed, Adnan Siraj Rakin, and Shaahin Angizi. Dnn-defender: An in-dram deep neural network defense mechanism for adversarial weight attack. *arXiv preprint arXiv:2305.08034*, 2023.

[92] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients.

[93] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. Hermes attack: Steal {DNN} models with lossless inference accuracy. In *USENIX Security*, 2021.