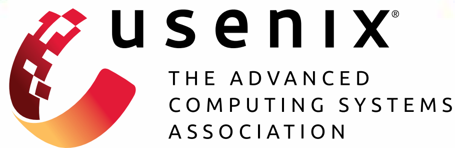

NeuroPots: Realtime Proactive Defense against Bit-Flip Attacks in Neural Networks

Qi Liu, *Lehigh University;* Jieming Yin, *Nanjing University of Posts and Telecommunications;* Wujie Wen, *Lehigh University;* Chengmo Yang, *University of Delaware;* Shi Sha, *Wilkes University*

https://www.usenix.org/conference/usenixsecurity23/presentation/liu-qi

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

NeuroPots: Realtime Proactive Defense against Bit-Flip Attacks in Neural Networks

Qi Liu
Lehigh University

Jieming Yin
Nanjing University of Posts and Telecommunications

Wujie Wen
Lehigh University

Chengmo Yang
University of Delaware

Shi Sha
Wilkes University

Abstract

Deep neural networks (DNNs) are becoming ubiquitous in various safety- and security-sensitive applications such as self-driving cars and financial systems. Recent studies revealed that bit-flip attacks (BFAs) can destroy DNNs' functionality via DRAM rowhammer –by precisely injecting a few bit-flips into the quantized model parameters, attackers can either degrade the model accuracy to random guessing, or misclassify certain inputs into a target class. BFAs can cause catastrophic consequences if left undetected. However, detecting BFAs is challenging because bit-flips can occur on any weights in a DNN model, leading to a large detection surface.

Unlike prior works that attempt to "patch" vulnerabilities of DNN models, our work is inspired by the idea of "honey-pot". Specifically, we propose a proactive defense concept named *NeuroPots*, which embeds a few "honey neurons" as crafted vulnerabilities into the DNN model to lure the attacker into injecting faults in them, thus making detection and model recovery efficient. We utilize *NeuroPots* to develop a trapdoor-enabled defense framework. We design a honey neuron selection strategy, and propose two methods for embedding trapdoors into the DNN model. Furthermore, since the majority of injected bit flips will concentrate in the trapdoors, we use a checksum-based detection approach to efficiently detect faults in them, and rescue the model accuracy by "refreshing" those faulty trapdoors. Our experiments show that trapdoor-enabled defense achieves high detection performance and effectively recovers a compromised model at a low cost across a variety of DNN models and datasets.

1 Introduction

Deep neural networks (DNNs) have achieved tremendous success in many real-world applications ranging from computer vision, speech recognition, disease diagnosis to safety- and security-critical autonomous vehicles, and banking systems [1, 2, 3, 4, 5, 6]. Unfortunately, recent studies have revealed that DNN inference execution on hardware engines such as GPUs, FPGAs, and ASICs, can be directly compromised by a variety of fault injection attacks [7]. In parallel with existing *data-centric* attacks [8, 9, 10, 11, 12, 13], these emerging *model-centric* attacks can also lead to attacker-desired persistent accuracy loss without altering the input.

Fault injection attacks disrupt DNN inference by manipulating the model's neuron activation, computed intermediate results, or weight parameters stored in buffers or memories, via active fault injection techniques such as laser beaming [14], row hammering [15, 16], or clock glitching [17]. Among these methodologies, the recently discovered **Bit-Flip Attack (BFA)** [16, 18] that exploits DRAM rowhammer vulnerability is considered to be one of the most destructive attacks. BFA injects faulty bits directly into memories that host weight parameters with DeepHammer [16], a type of rowhammer that can precisely flip a bit inside a DRAM page [19]. As demonstrated in real DNN testbeds, by flipping only 13 bits out of 1.2 billion bits of an 8-bit quantized DNN model, DeepHammer can degrade the model accuracy to random guessing within 66 seconds [16]. Moreover, a stealthy and probably more dangerous variant of BFAs called Targeted Bit-Flip Attack (T-BFA) was proposed [20]. T-BFA can go unnoticed for a long time as it only alters the classification of some inputs with a very marginal impact on others. Considering a resource-sharing environment where highly optimized DNN models are deployed to serve multiple users (e.g., machine-learning-as-a-service [5, 21]), BFAs can result in catastrophic consequences if left undetected.

Defending against BFAs, however, can be very challenging. BFAs leverage a progressive search algorithm to select and flip bits only in the most sensitive weights, and the locations of those bits are highly dependent on a very small portion of input data that attackers have. As a result, it is difficult to predict where BFAs will be landed and concentrate on those weights, as complex DNN models have too many "vulnerable points" to protect against (more details in Section 2.2). Defense solutions that passively detect faults of weights in pre-selected layers can be suboptimal [22]. To tackle this challenge, we propose a *proactive defense* con-

cept named "*NeuroPots*" (formally defined in Section 3.2). Inspired by the idea of honeypots, *NeuroPots* embeds a few specially designed "honey neurons" as vulnerabilities (i.e., trapdoors fully controlled by the defender) into the model to lure BFA's targeted chain of bit-flips. As honey neurons are more vulnerable than normal neurons, BFAs are highly likely to land on a few designated honey neurons. At runtime, faults in honey neurons can be efficiently detected using checksums, and the faulty weights can be recovered to the original value.

We comprehensively study *NeuroPots*'s theoretical and practical foundations of trapping bit-flips under real system constraints. Then based on *NeuroPots*, we design a trapdoor-enabled defense framework to detect and mitigate different basic and adaptive attacks. The evaluation shows that our design incurs **minimal impact on inference accuracy**, but is able to lure attackers to inject most bit-flips (if not all) into those honey neurons. By checking only a very small portion of model weights, attacks can be easily detected with a **close-to-zero false negative rate**. The discovered position information of the trapped bit-flips further offers a unique opportunity to **repair the model online at low cost** (e.g., 0.69ms time overhead and 99KB storage overhead for 14ms/22MB ResNet-34 in ImageNet).

To the best of our knowledge, this is the first work that adopts a HoneyPot-style proactive defense policy to offer extremely lightweight real-time mitigation on the challenging model-centric bit-flip attacks whose attack surface could be too large to be covered by existing solutions. We hope that our results could provide a new perspective for defending emerging attacks in deep learning and will enable more in-depth research along this direction.

2 Background and Motivation

In this section, we first provide a brief background on BFA, T-BFA, and DeepHammer. Then we explain why a proactive defense solution is crucial. Finally, we explain at a high level the design intuition behind *NeuroPots*.

2.1 Bit-Flip Attack

Bit-Flip Attack and Target BFA. The goal of BFA is to significantly degrade the model's accuracy by performing a minimum number of bit-flips. The objective function can be represented as: $\max_{\mathbf{B}} \mathcal{L}(f(x,\mathbf{B}),t)$, where $\mathbf{B}$ is the two's complement representation of quantized weights, t is the ground-truth target of input x, $\mathcal{L}$ denotes the loss function of the model. Unlike BFA, T-BFA aims to misclassify inputs from source category p into the target category q ($q \neq p$). Meanwhile, the remaining inputs will maintain their original categories (with no impact on their accuracy) to ensure attack stealthiness. T-BFA will minimize the following objective function:

$$\min_{\mathbf{B}} \mathcal{L}(f(x,\mathbf{B}),t_q)|x \in \mathbb{X}_p) + \mathcal{L}(f(x,\mathbf{B}),t)|x \notin \mathbb{X}_p) \quad (1)$$

Although BFA and T-BFA have different objectives, they both utilize a gradient-based Progressive Bit Search (PBS) to find the most vulnerable bits iteratively on quantized DNNs. PBS consists of two steps: intra-layer and cross-layer search. In the k-th iteration, the intra-layer search selects bits with top n gradient $\bigtriangledown_b \mathcal{L}$ as the most vulnerable bit candidates at layer l. For a q-bit quantized DNN, $\bigtriangledown_{\mathbf{b}} \mathcal{L}$ can be represented as:

$$\bigtriangledown_{\mathbf{b}} \mathcal{L} = [\frac{\partial \mathcal{L}}{\partial b_{N_q-1}}, \dots, \frac{\partial \mathcal{L}}{\partial b_0}] \quad (2)$$

Then, they apply a classical Fast Gradient Sign Method (FGSM) [8] algorithm from adversarial attack to the n selected vulnerable bit candidates, which can be presented as:

$$\mathbf{b}^* = \mathbf{b} \oplus \left(\mathbf{b} \oplus \left(\frac{\pm sign(\bigtriangledown_{\mathbf{b}} \mathcal{L}) + 1}{2}\right)\right) \quad (3)$$

Note, BFA uses +$sign(\bigtriangledown_{\mathbf{b}} \mathcal{L})$ to maximize objective function, while T-BFA applies -$sign(\bigtriangledown_{\mathbf{b}} \mathcal{L})$ to minimize its objective function. Then, they evaluate the loss increment as $\mathcal{L}_l^k$ for layer l. The same process will be performed in each layer to obtain loss set $\{\mathcal{L}_1^k, \dots, \mathcal{L}_l^k, \dots \mathcal{L}_L^k\}$. Then, the cross-layer algorithm identifies the j-th layer with maximum loss (i.e., j=$\arg\max_l \{\mathcal{L}_1^k, \dots, \mathcal{L}_l^k, \dots \mathcal{L}_L^k\}$) and flip the most vulnerable bit with largest gradient in j-th layer.

DeepHammer Attack (BFA via row-hammer). At the software-level, DeepHammer attack [16] optimizes BFA's vulnerable bits search algorithm. In the k-th iteration, after selecting n vulnerable bits by ranking gradients $\bigtriangledown_b \mathcal{L}$ like BFA, DeepHammer flips each bit respectively and generates a loss set as $\{\mathcal{L}_l^k\}_{i=1}^n$. The same process will be performed for each layer. As a result, the total candidate bits will be $n \times l$ and the corresponding loss set is $\{\mathcal{L}^k\}_{i=1}^{n \times l}$. DeepHammer selects the bit that has the maximum loss as the most vulnerable bit. At the hardware level, a new double-sided row-hammer attack using a targeted column-page-stripe data pattern is proposed to perform precise bit-flips in a DRAM page. Unlike the previous two ideal attacks, to achieve precise bit flipping, DeepHammer needs to obey a strict constraint, that is, **only one bit-flip per page**. In this work, we assume BFA and TBFA can flip any number of bits per DRAM page, while DeepHammer is constraint by flipping only one bit per page.

2.2 Motivation for Proactive Defense

To maximize the attack efficiency, BFAs always perform bit-flips on the most sensitive weights with the largest gradients. Because the gradients for weights highly rely on inputs, the generated bit-flip chain can be very different (i.e., different combinations of bits, weights, and layers) if attackers use different input data. Fig. 1(a) shows how bit-flips performed by BFAs change as input data alters. In this study, we follow the original setup of BFAs, first use three random seeds to pick three batches of input data, and then perform BFAs on

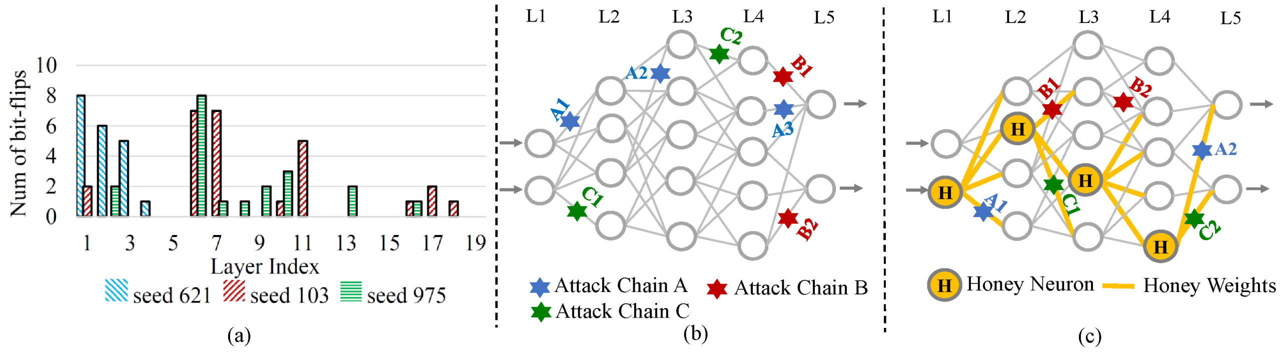

Figure 1: Comparison of the normal model and trapdoored model with *NeuroPots* under BFAs. (a) Layer-wise bit-flips distribution of 3 attack chains with different seeds for ResNet-20 (normal model). (b) A diagram of the position of bit-flips of 3 attack chains in a normal model. (c) A diagram of the position of bit-flips of 3 attack chains in a trappdoored model with *NeuroPots*.

each batch of input data individually. By only flipping 20 bits, BFAs can effectively degrade the model accuracy to random guesses for ResNet-20 on CIFAR-10. From the three trails of attack, we notice that although each trial focuses on flipping bits in just a few layers, none of the layers receives a bit-flip in all three trials. This study demonstrates that DNN models have many unpredictable natural “vulnerable points”, and hence defense solutions that rely on detecting faults of weights in a few pre-selected layers can be suboptimal. In other words, passively finding and examining a subset of neurons (or layers) that is more vulnerable to BFAs can easily lead to false negative given the limited coverage. Moreover, it fails when facing adaptive attackers who can circumvent these “fixed” vulnerable points and find another attack path.

The above observation motivates us to design a proactive defense mechanism. Instead of blindly analyzing which layers/weights the highly unpredictable BFAs will land on, we intentionally introduce vulnerabilities into the model to attract BFAs, such that the locations of faults become determined. If the crafted vulnerabilities are always easily discovered, we can ensure that attackers inject bit-flips into them even if using different data. Furthermore, as we only need to protect a few vulnerabilities, the detection overhead will be trivial even using precise but high-cost detection methods. More excitingly, if the vulnerabilities can trap most of the bit-flips, we can quickly recover model accuracy by “refreshing" faulty weights. To our best knowledge, no prior work can recover model accuracy to the original level at run time.

2.3 *NeuroPots* Design Intuition

Consider a scenario where, starting from a trained DNN model with a quantized parameter set $\mathbf{B}$ and any input x, the attacker searches for a universal adversarial binary weight-bit perturbation to induce a misclassification from the correct label y_x to any incorrect label other than y_x in BFA (or a target label $y_t \neq y_x$ in T-BFA). This is analogous to looking for a “shortcut” from the model to achieve the required systematical misclassification for any legitimate input x by flipping the least amount of bits in the model parameter set $\mathbf{B}$ (e.g., 13 bit-flips out of 1.2 billion bits [16]). Along these lines, *NeuroPots intentionally* disguise a few neurons as decoys and create shortcuts that are easier to locate and shorter than any natural weakness attackers are searching for. Once the model is trapdoored, an attacker will generate the adversarial chain of bit-flips along shortcuts produced by *NeuroPots*.

Fig. 1(b) and 1(c) respectively depict the distribution of bit-flips of three attack chains in the DNN model without and with *NeuroPots*. Without *NeuroPots*, the bit-flips of the three attack chains can appear in any weights across all layers. In Fig. 1(c), we embed one honey neuron in each layer to construct a trapdoored model. We can observe that 5 out of 6 bit-flips of the three attack chains appear in weights connected with the embedded honey neurons. Because at least one bit-flip appears in honey weights for each attack chain, we can therefore detect the attack accurately (we assume that the detection approach can correctly detect faults of honey weights). Moreover, all bit-flips of attack chains A and C are trapped by *NeuroPots*, so these two trials can achieve completed accuracy recovery by replacing faulty weights with their golden backup. Because honey weights are a tiny portion out of the entire weights in our experiments (e.g., $\sim 0.5\%$ in ResNet-34), the time and storage overhead for fault detection and model recovery will be very low.

3 Trapdoor Defense using *NeuroPots*

We utilize *NeuroPots* to design a trapdoor-enabled proactive defense framework. The key is to expand specific vulnerabilities in the model via designed honey neurons. Such intentional weaknesses we build into the DNN model (called *“trapdoored model”*) will shape and lure bit-flip attacks to make them easily detected and recovered at inference time.

3.1 Threat Model and Design Objectives

Threat Model. We adopt the white-box threat model assumption consistent with prior relevant works [16, 18], of which

the goal is to crush a well-trained quantized DNN model by flipping a minimum number of weight bits after model deployment. In particular, the attacker is assumed to possess complete knowledge of the victim trapdoored model's structure, weights, gradients, and partial knowledge of the training data. This can be obtained by ways such as side-channel attacks [23, 24]. The attacker's program co-locates with the victim model on resource-sharing platforms such as machine-learning-as-a-service with DRAM hosting quantized parameters. The attacker searches for the target chain of bit-flips via gradient-based BFA algorithm [18] and performs the DeepHammer attack [16] to precisely flip DRAM bits selected by the BFA algorithm. Under real system constraints, we assume DeepHammer only flips one bit per page. We also assume the attacker does not have access to proposed detector (e.g., checksum or bit-by-bit check used at run time to detect BFA or DeepHammer) and the small number of backup weights associated with *NeuroPots* for online model recovery (e.g., secured in the trusted execution environment (TEE) like Intel SGX [25] or ARM Trustzone [26, 27]). If ever compromised, trapdoor detection and recovery can be reset.

Adaptive Adversaries. We define three types of attackers based on their knowledge levels as follows. We evaluate the online detection and model recovery capabilities against all attackers in Section 6.

1. *Basic Attacker* is an entry-level attacker with no knowledge of the trapdoor defense. The attacker directly applies BFAs to the trapdoored model.
2. *Expert Attacker* is aware that the model is protected by trapdoors and detection will examine the flipped bits during inference. The attacker may also know some basic principles of trapdoor designs (e.g., algorithms for selecting and creating honey neurons) and try to leverage such knowledge to circumvent the defense. However, the attacker does not know the exact characteristics of trapdoor defense (e.g., the locations and number of honey neurons).
3. *Oracle Attacker* knows comprehensive details of trapdoor defense, for example, the design principles, the number and exact locations of embedded honey neurons, etc.

Defender's Capability. We consider two scenarios: 1) The defender has the ability to fine-tune the well-trained model using training data (often a few epochs), so as to quickly create trapdoored models. This is referred to as *retraining-based strategy* described in Section 4.1.2. 2) The defender does not have access to training hardware but a small number of testing data, and can directly build the trapdoor from the trained model without involving retraining. This is referred to as *one-shot strategy* described in Section 4.1.2).

Design Goals. We have three major design goals. First, the defense should detect adversarial bit-flips with very high precision and/or close-to-zero false negative rate under different levels of attackers; and recover model accuracy to almost the original level (e.g., $< 3\%$) for basic and expert attackers[1]. Second, the trapdoored model should not impact inference accuracy. Third, the defense should incur very low overhead, for example, a very small number of honey neurons for detection and negligible latency impact on normal inference.

3.2 Theoretical& Practical Basis of *NeuroPots*

In this section, we present formal theoretical principles as well as practical implementation principles of our trapdoor-enabled defense. These principles guarantee effectiveness in mitigating adversarial bit-flips. They also lay solid foundations for our detailed defense implementation in Section 4.

Theoretical Foundation of *NeuroPots*. To lure an attacker into flipping weight bits within our designed *NeuroPots*, we examine how the attacker uses a greedy strategy to identify the most vulnerable weight bits from the model. While the attacker's choice of a chain of bit-flips could be highly unpredictable (see Section 2.2), the first critical step is to identify a bit with the largest gradient w.r.t loss function from each layer based on a batch of training data. Then, the attacker compares the impact of each individual bit-flip to loss function and finally flips the most influential bit from one iteration to another. Intuitively, if we can intentionally magnify the gradient $|\bigtriangledown_W \mathcal{L}|$ of some selected weights, then we will be able to improve the chance of luring bit-flips to such weights and make the attacker's bit-flip chain predicable. Inspired by the above intuition, we further investigate the backpropagation rule to compute the gradient of loss function $\mathcal{L}$ w.r.t. weight W_{ij}^l that connects neuron j at layer l to neuron i at layer $l-1$:

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^l} = \begin{cases} g'(a_j^l) o_i^{l-1} \sum_{k=1}^{n^{l+1}} W_{jk}^{l+1} \delta_k^{l+1} & 1 \leq l < N \\ \delta_j^l o_i^{l-1} & l = N \end{cases} \tag{4}$$

where g is the activation function and δ_j^l is the error term of neuron j at layer l (i.e., $\delta_j^l = \frac{\partial \mathcal{L}}{\partial a_j^l}$, in which a_j^l is neuron j's output before passed to g at layer l). In a hidden layer (i.e., $1 \leq l < N$), the gradient magnitude of loss function $\mathcal{L}$ w.r.t. W_{ij}^l is determined by three key terms: $g'(a_j^l)$ — the derivative of activation function for neuron j at layer l; o_i^{l-1} — neuron i's activation value at layer $l-1$; and $\sum_{k=1}^{n^{l+1}} W_{jk}^{l+1} \delta_k^{l+1}$ — the weighted summation of error terms of all neurons at layer $l+1$. The first term generally equals to a constant (either 0 or 1) due to the widely adopted ReLU function in DNNs ($g(x) = x$ if $x \geq 0$, otherwise 0). Meanwhile, the last term cannot guarantee the increase of the absolute value of the summation, because weights and error terms could be negative or positive As a result, we focus on the second term o_i^{l-1}, which is the activation value of source neuron i at current layer $l-1$. *By just increasing o_i^{l-1}, theoretically the gradient*

[1]The online model repair is meaningful only if the recovered accuracy can be close to its original level. We do not consider such recovery for the oracle attacker due to a complete defense bypass.

magnitude of all weights (W_{ij}^l) starting from a source neuron i at the current layer to any destination neuron j at the next layer can be enlarged. This is also applicable to the output layer. In this way, we have the following formal definitions:

DEFINITION 1. *Any neuron (or feature map in a convolutional layer) with its activation value intentionally enlarged can be defined as a "honey neuron" (or "honey feature map"). All weights connected from the honey neuron to any neurons at the next layer are "honey weights" that can be used to trap malicious bit-flips from an attacker.*

DEFINITION 2. *"NeuroPots" consist of a set of honey neurons and the weights connected from each honey neuron.*

DEFINITION 3. *Any DNN model with "NeuroPots" embedded is defined as a trapdoored model.*

Practical Foundation of *NeuroPots*. Another important consideration of *NeuroPots* is to cover *scatted bit-flips* which could stem from an expert attacker under real system constraints (e.g., DeepHammer with one flippable bit per 4KB DRAM page). Fig. 2 depicts an example of mapping and storing honey neuron's connected weights in DRAM pages for a convolutional layer. In this example, we use a honey feature map (marked in yellow) instead of a single honey neuron, and its connected weights to the next layer are 2D filters in the same channel (marked in yellow) across all 3D kernels. Since the computation of every output feature map needs to read its corresponding 3D kernel from memory, the weights will be stored in the granularity of an entire 3D kernel for better access to a page. Given that honey weights in the yellow 2D filter appear in every 3D kernel, and the size of the total kernels in a convolutional layer (e.g., 256 3D kernels with $3 \times 3 \times 64$) is much larger than a DRAM page size (e.g., 4KB), honey weights will be stored across several pages to trap bit-flips, even if only a single honey feature map is embedded. This is also applicable to fully connected layers — the weight of a single honey neuron will be accessed whenever computing an output neuron of the next layer.

Why Focus on Neurons? We need to emphasize that *NeuroPots* focus on neurons/feature-maps instead of individual weights to create figurative holes. When attackers construct adversarial bit-flips against the model, they will have a high probability of falling into the trap. This is based on the fact that the number of neurons/feature-maps is usually much smaller than the weights in modern DNNs, and this can significantly reduce the attack surface and hence achieves efficient and effective detection and mitigation in real-time.

4 Detailed Defense Implementation

A useful defense should be able to mitigate different levels of attacks which are unpredictable but may occur in practical scenarios. Therefore, the design of trapdoor-enabled defense shall proactively consider defeating adversaries with different capabilities (see Section 3.1). To achieve this, we first introduce the two-step defense implementation: 1) offline trapdoored model construction; and 2) online adversarial bit-flip detection/mitigation. We then present the final defense recommendation against basic and expert attackers.

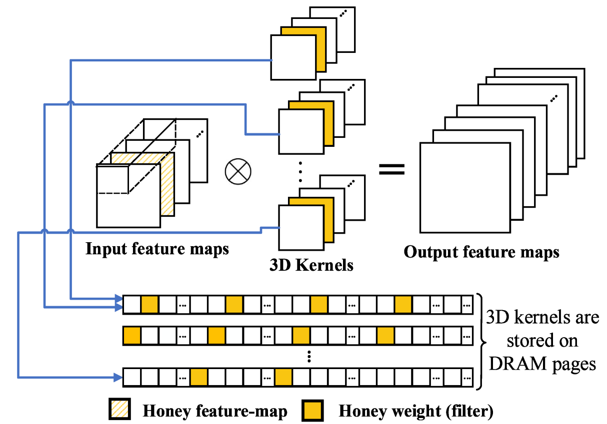

Figure 2: A conceptual view of mapping and storing honey weights to DRAM pages for a convolutional layer. Each DRAM page consists of multiple honey weights from the highlighted 2D filters, covering the scattered bit flips.

4.1 Step 1: Trapdoored Model Construction

Our first step is to construct a trapdoored model offline before model deployment, in which honey neurons and their associated weights are carefully designed and stealthily embedded into a trained model.

4.1.1 Honey Neuron Selection

Since the basic idea of trapdoor design is to intentionally make a small number of neurons more vulnerable, a simple approach would be to design a honey neuron selection strategy based on the ranking of neuron activation response. Intuitively, activating such already highly activated neurons is more likely to trap bit-flips produced by basic attackers, with less impact on inference accuracy. However, for skilled attackers who are aware that honey neurons are selected by ranking activation values, it may not work well since attackers may avoid flipping bits associated with those highly activated neurons to bypass the defense. Therefore, our solution needs to *enforce multiple levels of randomness in honey neuron selection*: 1) the positions of honey neurons (e.g., which layer and which neuron in a layer); 2) the activation rankings of honey neurons (e.g. ranking at different levels); and 3) the number of honey neurons. In this process, we also avoid selecting "dead" neurons that always produce zero activation.

We further treat fully connected layer (neuron) and convolutional layer (feature map) differently in honey neuron selection. In particular, for neurons in a *fully connected layer*,

we can randomly select the N neurons at a layer l. For neurons in a *convolutional layer*, considering that weights (or kernels) are shared by different neurons within the same output feature map, to simplify the design, we could treat an entire input feature map as a honey neuron, a.k.a honey feature map, and all its corresponding 2D filters in 3D kernels become honey weights (see Fig. 2). As a result, we randomly select the N feature-maps as honey feature maps at layer l.

4.1.2 Honey Neuron Embedding

Once honey neurons are identified, the next step is to embed them into a well-trained DNN model to create trapdoors. Meanwhile, trapdoor embedding should not degrade the model accuracy. To achieve the above goals, we need to solve the following optimization problem:

$$\min_{\theta} \mathcal{L}(f(x,\theta),y) + \alpha \cdot \sum_{l=1}^{N_h} \mathcal{L}_h(o^l,t^l) \tag{5}$$

where $\mathcal{L}_h$ is the loss function for embedding honey neurons, and α $(0 < \alpha < 1)$ is to balance model accuracy (first term) and trapdoor embedding (second term). N_h is the number of layers embedded with honey neurons. o^l is the honey neuron's activation at layer l, while t^l is its target activation value ($t^l \gg o^l$). In our implementation, we set the target t^l as γ times initial value of o^l (before trapdoor embedding), i.e., $t^l = \gamma \cdot o^l$ ($\gamma > 1$ is an expanding coefficient). To simply our design, we set the same γ_c for all convolutional layers and the same γ_f for all fully connected layers. To solve the above optimization problem, we propose two different methods: retraining-based embedding and one-shot embedding.

Method 1: Retraining-based Embedding (Fine-tuning). To optimize Eq. 5, a straightforward method is to fine-tune the original model using an optimization (training) algorithm. For trapdoor embedding, our goal is to minimize the difference between honey neuron activation and its target. We use Mean Square Error (MSE) as an extra loss term to achieve this goal:

$$\mathcal{L}_h(o^l,t^l) = \frac{1}{k}\sum_{i=1}^{k}(o_i^l - t_i^l)^2 \ \ (l \neq 0) \tag{6}$$

where k is the number of honey neurons, and t_i^l is the target (constant) activation value for honey neuron i at layer l. It is worth noting that honey feature maps embedding in convolutional layers will be conducted at the granularity of a single neuron of a feature map. Specifically, given a honey feature map o_i^l, we have $(o_i^l - t_i^l)^2 = \frac{1}{n}\sum_{j=1}^{n}(o_{ij}^l - t_{ij}^l)^2$, where n is the number of neurons of a certain feature map at layer l. Intuitively, we can also concentrate on a small portion of neurons (rather than all) of the feature map for reducing potential accuracy degradation. However, our experiment indicates there is not much difference in accuracy from acting on all neurons of the feature map when the number of honey feature maps is small. Thus, for design simplification, we will focus on all neurons of a feature map in real implementation.

We use ADAM [28] as our optimization algorithm to minimize Eq. 5 and Eq. 6. In general, we set a small value for α in Eq. 5 (e.g., 0.01), since the value of $\mathcal{L}$ is very small for a well-trained model, while the target activation value of honey neuron, namely t_i^l, is much larger than o_i^l at the beginning of the retraining process. Note that the retraining-based trapdoor can be applied to any layer except for the input layer (i.e., $l = 0$). This is because the input data (e.g., pixels or words) cannot be enlarged by fine-tuning parameters (e.g., weights).

Method 2: One-shot Embedding (Retraining-free). The retraining-based trapdoored model construction is relatively complex and costly, and it is only for defenders who have the ability to fine-tune models using a large amount of training data. The one-shot trapdoored model construction addresses those issues. Specifically, the one-shot strategy aims to quickly solve Eq. 5 by a set of simple computations with very low cost, using the following two steps. First, we directly enlarge the activation value of a honey neuron by multiplying γ, i.e., $o^l \leftarrow \gamma \cdot o^l$, to achieve trapdoor embedding (the second term of Eq. 5 will be 0). Second, we shrink all weights connected to the honey neuron by $\frac{1}{\gamma}$, so that the weighted contributions from this neuron to neurons in the next layer remain almost unchanged, hence minimizing the first term of Eq. 5. The specific one-shot processing can be formulated as:

$$o_i^{l+1} = \sum_{j=1}^{n^l} w_{ji}^l \cdot o_j^l = w_{0i}^l \cdot o_0^l + \cdots + (\frac{1}{\gamma} \cdot w_{hi}^l)(\gamma \cdot o_h^l) + \cdots \tag{7}$$

where o_h^l is the honey neuron at layer l, and w_{hi}^l is the associated honey weights. For a full-precision model, this one-shot processing will not bring any errors (we ignore bias here). For a quantized model, however, because we need to re-quantize the shrunk weights $\frac{1}{\gamma} \cdot w_{hi}^l$ to fixed-point numbers, this will bring a small amount of quantization error. A large γ could also lead the original weight to 0 (behaves like pruning) if the weight is small. However, we found that it only has a minor impact on model accuracy when embedding a few honey neurons. For the honey feature map, like retraining-based trapdoor embedding, the one-shot processing will be applied to all neurons of a feature map in a convolutional layer. Note that the one-shot strategy can be applied to any layer (including the input layer) because it will directly change the activation value of honey neurons.

Impact of Trapdoor on Model Output. After finding the most sensitive bit, the attacker will further measure the impact of bit flipping on the loss function to verify if the flipping can degrade model accuracy. Thus, to trap bit-flips efficiently, we need to ensure that our trapdoor can cause a more considerable change in model output when bit-flips are injected. For a normal neuron, its impact on next layer's output under BFAs can be expressed as $o^{l+1} = (w + \Delta w) \cdot o^l$, where Δw is the weight distortion caused by bit-flips. In comparison, taking the one-shot trapdoor as an example, a honey neuron's impact

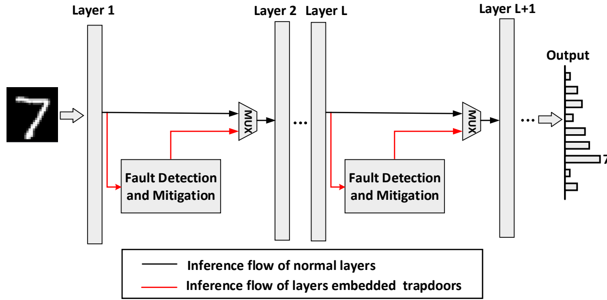

Figure 3: Trapdoor-based fault detection and mitigation.

to the next layer is represented as:

$$o^{l+1} = (\frac{1}{\gamma} \cdot w + \Delta w) \cdot \gamma \cdot o^l = (w + \Delta w) \cdot o^l + (\gamma - 1) \cdot \Delta w \cdot o^l \quad (8)$$

We can see o^{l+1} increases by $(\gamma - 1) \cdot \Delta w \cdot o^l$ compared to the normal neuron. Moreover, the attacker is more likely to flip the most significant bit of weights to produce a large perturbation Δw, so the impact of the honey neuron to o^{l+1} will be more significant, especially for a large γ. Such change will propagate and accumulate to the following layers, causing a dramatic change to the model output. Therefore, our trapdoor can trap bit-flips efficiently.

Activation Ranking Obfuscation. While using a larger expanding coefficient–γ to intentionally activate honey neurons significantly can achieve a better bit-flip trapping efficiency in basic attackers, expert attackers may still use such information to bypass the defense. Therefore, we propose to moderately enlarge the activation of honey neurons using a smaller expanding coefficient (e.g., $\gamma = 2$) during embedding. In this way, honey neurons' ranking can be moved up slightly and distributed across all ranking positions in a layer, increasing the stealthiness. Even if attackers bypass higher-ranked ones, those with a lower ranking can still trap bit-flips.

4.2 Step 2: Online Detection/Model Recovery

After constructing the trapdoored model offline, the next step is to take advantage of it for online fault detection and model recovery. As Fig. 3 shows, we add fault detection and mitigation into layers embedded with the trapdoor, while other layers simply follow the original inference flow. Because honey weights are a tiny portion of the DNN model, the fault detection and recovery overhead will be very low. Therefore, our defense has a trivial impact on the original inference. We evaluate the inference time and storage overhead in Section 6.

4.2.1 Fault Detection

Because our trapdoor can significantly reduce the detection surface, lightweight fault detection could be easily realized, for example, via simple checksum or distance-based comparison. As an example, we develop a simple checksum mechanism to detect faults on honey weights. If the sum of the current honey weights is different from the original sum, the model is then considered to be tampered. In particular, the method takes advantage of Multiply-Accumulate (MAC) operations commonly available in DNN hardware, and thus can eliminate additional hardware support for detection. Also, instead of checking the sum of all honey weights, it checks the sum of honey weights at the granularity of filters to improve detection performance. Fig. 4 illustrates the procedure of utilizing convolution (or MAC) operations to compute the sum of honey weights without additional hardware support. Specifically, we set the honey feature map into 1 and others to 0. After conducting convolution operations with a 3D kernel, the first neuron's value in the output feature map will be the sum of the honey weights (filter). For fully connected layers, computation of neurons output can be treated as 1×1 convolution operation, so we can also use similar operations to compute the sum of honey weights.

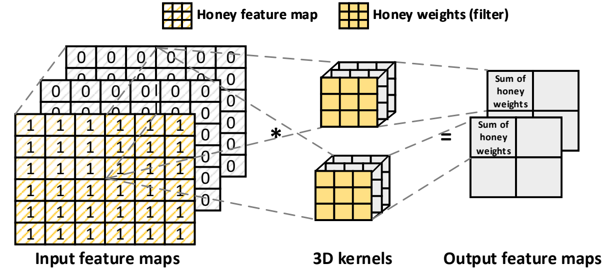

Figure 4: Example of computing the filter-wise checksum of honey weights (filter) with convolution (MAC) operations.

4.2.2 Run-time Model Recovery

After building the trapdoored model, the defender will need to store a copy of clean honey weights in a secured zone (e.g., trusted execution environment (TEE) like Intel SGX [25] or ARM Trustzone [26, 27]). Once faults are detected, we can directly replace faulty honey weights with the clean copy at run-time. In particular, to fit our filter-wise checksum detection, the level of granularity for fault mitigation is set as filter. Because we only need to store/recover a very small amount of weights, the storage and time overheads are extremely low. Furthermore, since our trapdoor can trap most of the bit-flips and mitigate the faults, the recovered model will be very close to the original model and hence the inference accuracy. It is worth mentioning that the recovered model is still a trapdoored model, so we do not need to repeat the trapdoor embedding process.

4.3 Putting It All Together For Final Defense

We build our final trapdoor-enabled defense to effectively mitigate different basic and expert attackers by combining all aforementioned techniques via the following manner: 1) random honey neuron selection (position), neuron activation

ranking obfuscation (via tuning expanding coefficient γ in embedding) and layer-wise coverage (via adjusting honey neuron number); 2) filter-wise fine-grained fault detection and model recovery in real time. In the following sections, we use such a combination as the final recommended defense for evaluation and discussion.

5 Adaptive Attack Design

Beyond the basic adversary, any meaningful defense must withstand countermeasures from expert attackers who acknowledge the defense. In Section 3.1, we define attackers with different abilities to comprehensively evaluate trapdoor-enabled defense. Particularly, attackers without any knowledge of defense (original BFA, T-BFA and DeepHammer Attack in Section 2.1), are referred as *basic attackers*.

In this section, we design *expert attackers* that are aware that the target model has been embedded with trapdoors, and know the basic principles of trapdoor design but no specific details. We present multiple *expert attackers* separated into two broad categories. First, we consider the *bit (weight)-level bypass* approach (**BBA**) by crafting an adversarial bit-flip chain from weight bits with lower gradients in the trapdoor protected model, and then use it to attack this model. The purpose is to avoid choosing an individual (weight) bit from established *NeuroPots* in the model that often has a higher gradient. Second, we consider *neuron-level bypass* approaches (**NBA**) that focus on finding bit-flip chains that cause misclassification while avoiding using all weight bits possibly associated with honey neurons.

5.1 Bit-level Bypass Attack (BBA)

To achieve the highest attack efficiency, a greedy non-expert attacker searches the more vulnerable bits of the model to generate an attack that would lead to the least number of bit-flips. Our trapdoor-enabled defense takes advantage of such an attack pattern to set a few disguised vulnerable traps, e.g., weight bits with intentionally enlarged gradients located in *NeuroPots*, to lure the attacker's bit selection. Therefore, an expert attacker who knows that a target model contains such trapdoors may simply lower the objective by picking weight bits whose gradients are not among the top list, to circumvent pre-defined *NeuroPots*. Accordingly, the gradient-based bit ranking search algorithm of this adaptive BFA can be modified as follows:

$$
\begin{aligned}
b_i^{n-1} &= Top_p|\bigtriangledown_{W_i^{n-1}} \mathcal{L}(f(x,\{W_i^{n-1}\}_{i=1}^{l}))| \\
\Longrightarrow b_i^{n-1} &= Top_p\left(Low_t|\bigtriangledown_{W_i^{n-1}} \mathcal{L}(f(x,\{W_i^{n-1}\}_{i=1}^{l}))|\right)
\end{aligned}
\tag{9}
$$

where $t \in (0, 100\%)$ is an adjustable threshold. The algorithm iteratively searches the most vulnerable p bits only from a subset of weights–$t\%$ of weights with lower gradients in a layer l, and excludes the rest (top) $(1-t\%)$ of weights with higher gradients that may contain trapdoors.

5.2 Neuron-level Bypass Attacks

Instead of examining individual weight bit, more advanced expert attackers would also consider defeating trapdoor by first speculating honey neurons, and then avoiding injecting bits in all their associated weights. Such neuron- (or *NeuroPots*) level bypass could potentially lead to better efficiency for expert attacks, since all weights associated with identified neurons will not be selected for bit flipping.

5.2.1 Ranking-based Neuron Bypass (R-NBA)

To trap more bit-flips, honey neurons generally need to have large activations. Therefore, expert attackers could utilize this property to identify honey neurons and bypass trapdoors. Specifically, the attacker may rank activation magnitude of neurons or feature maps (i.e., $|o|$ or $\sum_i^n |o_i|$), then *bypass* all weights associated with the neurons with $top_{k_b}|o|$ or $top_{k_b}\sum_i^n |o_i|$ (e.g., k_b = 10%) in certain layers. We name this attack as **R-NBA-b**. On the other hand, if expert attackers are aware that our trapdoor obfuscates the activation ranking of honey neurons, they may further upgrade their attacks by deliberately *attacking* the top-ranked neurons only to craft the attack, since these neurons may not be honey neurons. This attack can potentially generate a stealthy attack chain that would need much fewer bit-flips, thereby lowering the possibility of being detected by the trapdoor. This variant is called **R-NBA-a**.

5.2.2 Frequency-based Neuron Bypass (F-NBA)

Another approach to precisely detect and bypass honey neurons is to leverage the fact that, statistically, honey neurons are likely to receive more bit-flips than normal neurons. In this case, expert attacker can perform BFAs multiple times using different input data, during which he or she counts the number of bit-flips in weights associated with each neuron/feature-map. A neuron whose weights receive more bit-flips is likely to be a honey neuron, and all weights associated with this neuron can be excluded from the bit-flip search process. We name such a bypass attack as **F-NBA**. Specifically, this bit-flip frequency detection includes two stages: 1) Testing (detecting) stage –we perform the attack algorithm using different training/testing data m times to generate a honey neuron set $\{h\}$. In each testing, the number of bit-flips in weights associated with each neuron n_i will be counted. If the number of bit-flips on neuron n_i is larger than a threshold T (e.g., $T = 2$), then n_i is considered to be a honey neuron. We then add neuron n_i into the honey neuron set $\{h\}$ (i.e., $\{h\} \leftarrow \{h\} \cup n_i$). 2) Attacking (bypassing) stage–during attack we bypass all weights associated with the honey neurons obtained from the

last step $n_i \in \{h\}$. Generally, it is difficult for an attacker to obtain enough data to generate different bit-flip chains to test the target model many times. Here we make an aggressive assumption for the purpose of evaluation.

6 Evaluation

In this section, we evaluate the performance of our trapdoor-enabled defense against *basic adversary* and *expert adversary* as described in Section 3.1. Specifically, our evaluation answers the following questions:

1. How does *NeuroPots* impact the model accuracy?
2. How is the detection and mitigation performance of our defense framework against basic and adaptive BFAs?
3. How much time and storage overhead does our fault detection and model recovery framework introduce?

6.1 Experimental Setup

We use PyTorch as our implementation framework. All simulations are conducted in a workstation with one AMD Ryzen Thread ripper 2990WX 32-Core Processor and four NVIDIA GeForce RTX 2080Ti GPUs. Our DRAM is 32GB (X64, DR) 260-Pin DDR4 SODIMM and consists of 16 internal banks.

6.1.1 Datasets and DNN Structures

We evaluate the effectiveness of our trapdoor-enabled defense framework with CIFAR-10 [29] and ImageNet [30] for image classification and Google Speech Command [31] for speech recognition datasets. Specifically, CIFAR-10 consists of 60K of 32×32 colored images from 10 classes, in which 50K and 10K images are used for training and testing, respectively. ImageNet contains 256×256 colored images from 1000 classes, in which 1.3M, 100K and 50K images are used for training, testing, and validation, respectively. Google Speech Command includes 65,000 one-second-long clips from 12 classes (10 commands and 2 additional special labels), each of which has 30 different words.

We evaluate our trapdoor-enabled defense across different DNN structures. In particular, ResNet-20 [32] and VGG-16 [33] are evaluated on CIFAR-10, while the evaluations of AlexNet [34], ResNet-34, and a memory-efficient MobileNet-V2 [35] are conducted on the complex ImageNet. The Google Speech Command dataset is used to evaluate WideResNet-28 [36]. The weights of all these DNNs are quantized to 8-bit. The accuracy of well-trained quantized models are shown in Table 1 (see w/o trapdoor column).

6.1.2 Attack Configuration and Fault Models

Basic Adversary. For basic adversary evaluation, we use three types of original bit-flip attacks: BFA, T-BFA, and DeepHammer. The goal of BFA and DeepHammer is to degrade model accuracy to random guesses (i.e., 10%, 0.1%, 8.33% for CIFAR-10, ImageNet, Google Speech Commands, respectively). In comparison, T-BFA only misclassifies inputs from source class p to a target class q while maintaining accuracy for all inputs from other classes j. We randomly select a source and a target class in our experiment. In terms of hardware constraints of bit-flips in DRAM, we assume BFA and T-BFA can inject bit-flips into any weights of the model. In contrast, as a real bit-flip attack via row-hammer, DeepHammer only flips one bit per DRAM page. For each type of BFAs, we repeat the above attack configuration 50 times to generate 50 bit-flip chains (i.e., 50 fault models) using different random seeds for each DNN.

Expert Adversary. We consider that attackers can integrate the defense bypass techniques described in Section 5 with realistic DeepHammer to develop four adaptive DeepHammer attacks: BBA, R-NBA-b, R-NBA-a, and F-NBA. Without loss of generality, we set the configurations for these attacks as follows: $t = 98\%$ for BBA (i.e., bypass weight bits with top 2% gradient), $k_b = 5\%$ for R-NBA-b and R-NBA-a (i.e., bypass/only-attack 5% top-ranked neurons), $T = 2$ and test times is 50 for F-NBA (i.e., run the attack algorithm 50 times with different seeds in the targeted model and bypass all neurons that are attacked more than two times in any trials). For simplicity, we focus on discussing these attacks under a given defense budget (see defense budget setting discussion in Section 6.1.3). We leave more evaluations under different adaptive attack parameter settings in Appendix C.

The detailed configurations for three basic BFAs and four adaptive attacks in all DNN models, including the number of needed bit-flips, are presented in Table 8 of Appendix A.

6.1.3 Guidelines of Setting Key Parameters in Defense

We follow the procedure in Section 4.1 to create honey neurons for defense. In particular, we randomly select honey neurons and apply activation ranking obfuscation within a layer. To increase their stealthiness and improve the trapping rate, honey neurons are distributed across more layers. To properly set the three key parameters of this process: 1) number of honey neurons per layer N, 2) expanding coefficient γ, and 3) number of selected layers N_l, we follow the general guidelines described below. A detailed ablation study is provided in Section 7.1.

To choose an appropriate N for different models, we empirically set the honey neuron number per layer–$N = max(2, \lceil \frac{p\% \times N_m}{N_l} \rceil)$, where p is a given budget (percentage) of honey neurons that can be used in defense, N_m is the total number of neurons in the model and N_l is the number of selected layers. These parameters determine the defense performance and time/storage overhead. In our evaluation, we consider the following typical defense budget: the defense incurred time overhead is no more than 10% of a single inference time in any model, and the proportion of all se-

Table 1: The accuracy (%) of original and trapdoored models (w/o and w/ trapdoor) on various datasets and DNN structures.

Dataset	Structure	w/o trapdoor	w/ trapdoor retraining	w/ trapdoor one-shot
CIFAR-10	ResNet-20	90.34	89.34	90.24
	VGG-16	92.07	91.3	91.82
ImageNet	AlexNet	57.47	56.16	56.04
	ResNet-34	72.56	71.21	71.05
	MobileNet-V2	71.89	70.16	69.96
Google Speech Command	WideResNet-28	97.73	96.58	97.12

lected neurons in any model is $< 1\%$, except for ResNet-20 where 2.8% are honey neurons because it only contains ~ 700 neurons/feature-maps). We set N_l to 10 for all DNN models. Note that, AlexNet only has 8 layers, so we embed honey neurons to all layers of AlexNet. For honey neuron embedding, we use a small expanding coefficient (e.g., $\gamma = 2$) to ensure stealthiness and maintain original accuracy for all models. To find a suitable expanding coefficient for each model, our practice is to start with a relatively larger value (e.g., $\gamma = 4$). If it incurs a noticeable impact on model accuracy, or all honey neurons are top-ranked, we then gradually reduce its value until not all honey neurons are top-ranked and the accuracy drop is acceptable (e.g., less than 3%). The detailed defense parameter settings are listed in Table 7 of Appendix B.

6.1.4 Evaluation Metrics

We focus on three evaluation metrics: 1) **detection rate** is defined as the ratio between the number of models correctly identified as malicious and the total number of malicious models. A higher detection rate indicates better defense efficiency; 2) **mitigation success rate** is calculated as the number of models with accuracy recovered close to original level after defense divided by the total number of malicious models. Here we define a successful mitigation empirically if the accuracy difference between the recovered model and clean model (without trapdoor) is less than 3%; 3) **trapping rate** is the proportion of trapped bits out of the total number of bit-flips injected by attackers. Intuitively, a higher trapping rate indicates a better mitigation success rate.

6.2 Results and Analysis

6.2.1 Impact on Performance

If not properly selected and embedded, honey neurons can cause model accuracy drop. We first evaluate how *NeuroPots* impacts the model accuracy. Table 1 lists the accuracy of the original model (w/o trapdoor) and the trapdoored model (w/ trapdoor) in different DNN structures. We can observe that both trapdoor embedding techniques (see Section 4.1) only cause marginal accuracy drop for all DNN structures on different datasets compared to the clean models. For example, the retraining and one-shot trapdoored models of VGG-16 can achieve 91.3% and 91.82% accuracy, which are only 0.77% and 0.25% lower than their respective clean models.

6.2.2 Detection and Mitigation Effectiveness

Trapping Efficiency. Table 2 shows that our defense can trap most bit-flips in the attack chain based on three bit-flip chains generated by BFA on the retraining-based trapdoored model of VGG-16. For example, our model captures 10 out of 13 bit-flips in the first attack chain and achieves a 76.92% trapping rate. Although our defense might not trap all faults from attackers, the model's accuracy can still be recovered to the original level in the fault mitigation stage. This is because our defense always traps the most destructive bit-flips of the attack chain to interrupt the critical "attack path". For example, in Table 2, the first three bit-flips are trapped by our defense in all three attack chains. The attackers always search for the most vulnerable bits of current model iteratively. For each iteration, the current most vulnerable bit is on top of the prior bit-flips to maximize accuracy drop. If we can recover the first few critical bit-flips, the remaining bit-flips are less important.

Detection and Mitigation Performance. As Table 3 shows, our trapdoor-enabled defense can achieve a very high detection rate (100% in most cases) for three types of basic attacks and four types of adaptive attacks in all DNN models.

Table 3 also reports the mitigation success rate in different DNN models. On the one hand, for three types of basic attacks, our defense achieves a considerably high mitigation success rate in all DNN models. This indicates that most faulty models' accuracy are recovered to the original level at run-time. In particular, our defense achieves a similar mitigation success rate for BFA and T-BFA (90.33% vs. 90% on average) but a slightly worse performance for DeepHammer in most cases (85.33% on average). This is because DeepHammer has the one-bit-flip-per-page constraint, and therefore, the injected bit-flips are more scattered and a lower portion of them is trapped. On the other hand, for four types of adaptive attacks, our defense only achieves slightly worse mitigation effectiveness than the basic version (i.e., Deep Hammer) due to the randomness of selecting honey neurons. Such minor performance degradation means that these strong adaptive attacks have a very small chance to bypass a few honey weights/neurons.

6.2.3 Time and Storage Overhead

One of our design goals is to minimize the overhead of the trapdoor-enabled defense model. Table 4 shows the time spent on a model inference with and without the trapdoor, as well as the additional storage overhead. For time cost, our defense only takes slightly more time than the baseline due to involving fault detection and mitigation in the inference pipeline. For example, in VGG-16, it costs an additional 0.07ms (total 4.96ms) compared to the baseline (4.89ms). For networks with convolutional layers only (e.g. ResNet-20), the time over-

Table 2: Example bit-flip attack chains generated on retraining-based trapdoored VGG-16 (bold denotes trapped bits by defense).

# of bit-flips	Attack chain (layer number, in-layer offset of weight)	Trapping rate (%)	Accuracy (%)	Accuracy (%) after fault mitigation
13	**(3,36441)**->**(2,9913)**->**(2,9916)**->(1,363)->**(16,1771)**->**(16,3587)**->**(4,77753)** ->**(16,1984)**->(5,208)->**(16,3312)**->(7,226285)->**(4,2873)**->**(4,98494)**	76.92	10.65	90.31
6	**(3,36446)**->**(2,9913)**->**(4,98494)**->**(4,144574)**->**(16,1771)**->**(16,3587)**	100	10.72	91.3
10	**(2,21432)**->**(3,36447)**->**(4,98490)**->**(2,9913)**->(6,240883)->**(16,1771)** ->(5,243840) ->(5,204670)->**(16,3587)**->(5,48002)	60	10.84	91.34

Table 3: Detection/mitigation rate of trapdoor design (one-shot) on different models and datasets under basic and adaptive attacks.

Attack Type	Attack	CIFAR-10			ImageNet		Google Speech Command	AVG
		VGG-16	ResNet-20	AlexNet	ResNet-34	MobileNet-V2	WideResNet-28	
Detection Rate (%)								
Basic	BFA	100	100	100	100	100	100	100
	T-BFA	100	100	100	100	100	100	100
	DeepHammer	100	100	100	100	100	100	100
Adaptive	BBA	100	100	100	100	100	100	100
	R-NBA-b	100	100	100	100	100	100	100
	R-NBA-a	100	100	100	100	100	100	100
	F-NBA	100	100	100	100	100	100	100
Mitigation Success Rate (%)								
Basic	BFA	98	96	88	90	84	86	90.33
	T-BFA	98	94	90	92	80	86	90
	DeepHammer	92	90	84	86	78	82	85.33
Adaptive	BBA	88	76	80	64	76	74	76.33
	R-NBA-b	86	70	78	68	74	72	74.67
	R-NBA-a	90	86	80	84	78	76	82.33
	F-NBA	90	80	82	82	76	74	80.67

head is slightly increased but is $< 10\%$ due to having more honey feature maps. For storage cost, it requires almost negligible extra memory (e.g., 99KB or 0.5% in ResNet-34), to store a small number of clean honey weights for all DNN models. Overall, compared to traditional passive defense approaches, our proactive approach can effectively reduce the number of weights needed for fault detection and mitigation under BFAs, and incurs very low time and storage overhead.

7 Discussion

7.1 Ablation Study

In Section 6.1.3 we present general guidelines for selecting three key hyper-parameters for *NeuroPots*-based trapdoor defense. Without loss of generality, in this section we analyze how these parameters impact our model performance using ResNet-20 and CIFAR-10 dataset.

Impact of Expanding Coefficient γ. Fig. 5(a) shows how expanding coefficient impacts the trapping effectiveness under basic and adaptive attacks. For most of the attacks (e.g., Deep Hammer and BBA), a larger γ could significantly increase the trapping effectiveness. However, we can also observe that a larger γ decreases the trapping rate for R-NBA-b. This is because the weight gradients of honey neurons increases as γ grows. Once the gradients become large enough, these honey neurons will fall into the higher-ranked category, which R-NBA-b can effectively bypass. Therefore, to ensure a decent detection and mitigation performance for all attacks, we need to choose an appropriate γ (generally less than 3).

Impact of Number of Honey Neurons Per Layer N. In Fig. 5(b), we observe that more honey neurons in each selected layer improves the trapping rate for all attacks. However, having too many honey neurons suffers from higher time and storage overhead. For example, when $N = 5$, storing honey weights brings 3.38% extra storage overhead. Therefore, an appropriate number of honey neurons in each layer is essential to balance trapping effectiveness and overhead.

Impact of Number of Selected Layers N_l (or Honey Neuron Distribution). Fig. 5(c) shows how N_l impacts the trapping rate. In this experiment, we keep the total number of honey neurons unchanged (i.e., 20 honey neurons/feature-maps) and spread honey neurons evenly across each selected layer. For example, if the number of selected layers is 4, we select 5 honey neurons in each layer for the first 4 layers in the model. We found that distributing honey neurons across more layers can increase trapping rates for most attacks. However, for R-NBA-b, we observe that too scattered honey neurons could hurt trapping effectiveness (e.g., when the number of selected layers is larger than 10). This is because when N_l becomes too large, some layers may have only one honey neuron, and therefore R-NBA-b can bypass it so that the entire layer becomes unprotected. Therefore, we use at least 2 honey neurons in each selected layer with an appropriate number of selected layers (e.g., 10 layers) to ensure trapping effectiveness for all attacks.

Table 4: Time and storage overhead of our trapdoor-enabled defense on different DNN structures and datasets.

	Inference flow	CIFAR-10		ImageNet			Google Speech Command
		ResNet-20	VGG-16	AlexNet	ResNet-34	MobileNet-V2	WideResNet-28
Time Cost (ms)	w/o trapdoor (baseline)	1.13	4.89	48.22	13.45	31.98	12.19
	w/ trapdoor	+0.11 (9.7%)	+0.07 (1.4%)	+0.87 (1.8%)	+0.69 (5.1%)	+0.34 (1.1%)	+0.55 (4.5%)
Storage Cost (MB)	w/o trapdoor (baseline)	0.27	34	61	22	3.5	36
	w/ trapdoor	+0.0037 (1.3%)	+0.12 (0.4%)	+0.57 (0.9%)	+0.099 (0.5%)	+0.0038 (0.1%)	+0.33 (0.9%)

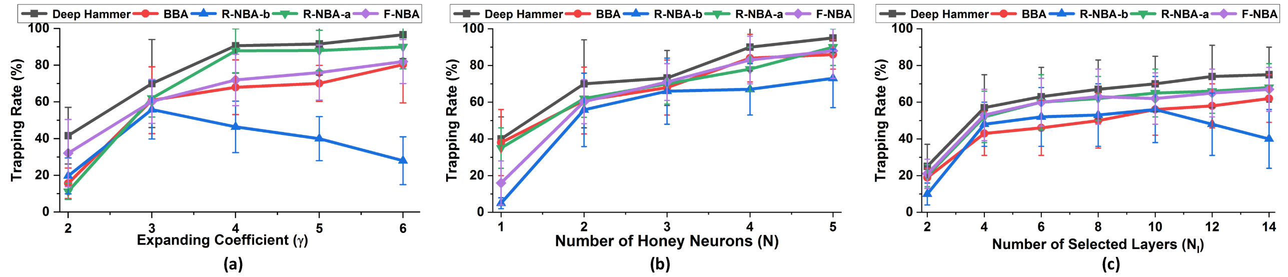

Figure 5: The impact of expanding coefficient γ (a), number of honey neurons per layer N (b), and number of selected layers N_l (c) on trapping rates.

Table 5: The trapping rate, detection rate and mitigation rate in different DRAM page sizes under DeepHammer attack.

Size	Trapping rate (%)	Detection rate (%)	Mitigation Success rate (%)
4K	72.3±8.5	100	92
8K	71.5±8.4	100	94
16K	69.5±7.9	100	92
32K	68.2±8.6	100	92
64K	68.9±8.2	100	92

7.2 Comparison and Discussion

Influence of Scattered Flips. We now discuss the impact of scattered bit-flips on defense performance. Keeping the one-bit-per-page constraint unchanged, we vary the page size to control the degree of scattering bit-flips in Deep Hammer. The default DRAM page size in Deep Hammer is 4K. As Table 5 shows, the detection rate remains 100% for all page sizes. The trapping rate of trapdoor defense remains stable beyond 4K page size. Similarly behavior is observed for mitigation success rate. As the page size grows, each page will contain more honey neurons. Although bit-flips becomes more scattered in larger pages, having more honey neurons in each page can effectively trap the bit-flips and stabilize the trapping rate (and hence mitigation rate). Overall, more scattered bit-flips only have a limited impact on our defense.

Comparison to Natural "Neural Pots". We inspect the relationship between natural "neural pots" (neurons with naturally very high activation with no additional embedding operation) and *NeuroPots*. To make a fair comparison, for natural "neural pots", we use the same configuration as *NeuroPots* (i.e., $N = 2$, $N_l = 10$ in ResNet-20). As Fig. 6 shows, the trapping effectiveness of natural "neural pots" is much lower than *NeuroPots*, especially for adaptive attacks such as BBA. This is because in natural "neural pots", without the embedding operation, there is no guarantee that the selected highly activated neurons remain highly activated for all inputs. Additionally, in some trials, those highly activated neurons can be bypassed by well-designed adaptive attacks. Therefore, the stealthy honey neuron embedding in *NeuroPots* is necessary.

Comparison to Random Neuron Monitoring and Runtime Analysis. A simple baseline defense could be monitoring neurons randomly to detect attacks. Fig. 7 compares our trapdoor defense with this random neuron monitoring defense. To achieve a relatively high detection rate (e.g., $> 90\%$), this baseline needs to monitor 50% of neurons at least, resulting in a high time overhead. For example, an extra 1.82ms is needed for an inference that normally takes 1.13ms to monitor 50% of neurons, totaling 2.95ms or 2.6× that original inference time–1.13ms). Compared to random neuron monitoring, trapdoor defense only needs to monitor 2.8% of neurons with 0.11ms additional time (e.g., total 1.24ms, 9.7% of the inference time) to achieve a 100% detection rate.

Comparison to Existing Detection Methods. One of the key capabilities of *NeuroPots* is to detect BFAs accurately in real-time at extremely low cost, hence we also experimentally compare trapdoor defense with detection-based defense. Since existing detection can be roughly divided into two categories: 1) machine learning-based–training a simple model for fault detection [22, 37]; 2) signature-based–embed unique signatures, e.g., watermark or checksum, into the original model and then compute and compare signatures on the fly to detect faults [38, 39, 40]. Without loss of generality, we choose two representative solutions in the respective category for the comparison: *weight encoding detection (WED)* [22] and *DeepAttest (a.k.a., Fingerprint)* [38]. In particular, WED applies weight sensitivity analysis to pick weights from a few most sensitive layers to train a single-layer perception to generate the detecting secret-key, while Fingerprint utilizes a shredder storage format to randomly select weights to embed fingerprint. Then the detecting secret-key or fingerprint extracted and computed from protected weights are compared

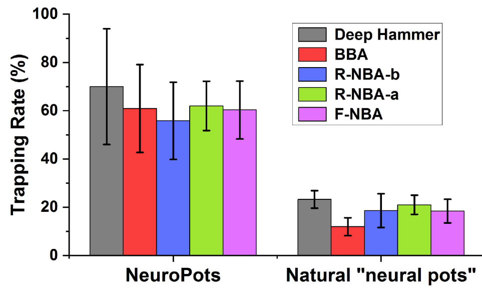

Figure 6: The comparison of trapping rate between our *NeuroPots* and natural "neural pots".

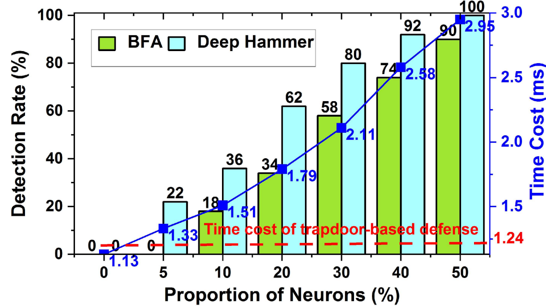

Figure 7: The detection rate and time cost by random neuron monitoring. Time cost of monitoring 0% neuron is the original inference time (1.13 ms).

with the golden ones for fault detection at the online stage. Since BFAs only change a few weights, it is nearly impossible to capture faulty weights using random weight selection. Thus, to construct a fair baseline and lower the detection overhead, we also apply WED's sensitive weight selection to Fingerprint. It is worth noting that neither Fingerprint nor WED offers fault mitigation, so we only compare detection efficiency. As Fig. 8 (a) shows, *NeuroPots* achieves a better detection rate than WED and Fingerprint. Moreover, as Fig. 8 (b) shows, *NeuroPots* incurs much lower time cost than the two baselines in all models due to significantly reduced attack detection surface (e.g., $90 \sim 100\times$ for AlexNet).

7.3 Defense Against Oracle Attack

An oracle attacker has all details about the trapdoor, including algorithm, the number and exact locations of honey neurons, such that the attacker can entirely circumvent *NeuroPots* and crush the trapdoor defense. We use this strongest attack which is often impractical, to test the upper bound of our defense. We design a hybrid countermeasure to ensure that the oracle attacker cannot circumvent *NeuroPots* without spending an unreasonable amount of effort. So far, we have explained how *NeuroPots* can be used to craft honey neurons. In the hybrid countermeasure described below, we also use *NeuroPots* to enhance normal neurons that are vulnerable to BFAs and make them more robust. We name such neurons as *enhanced*

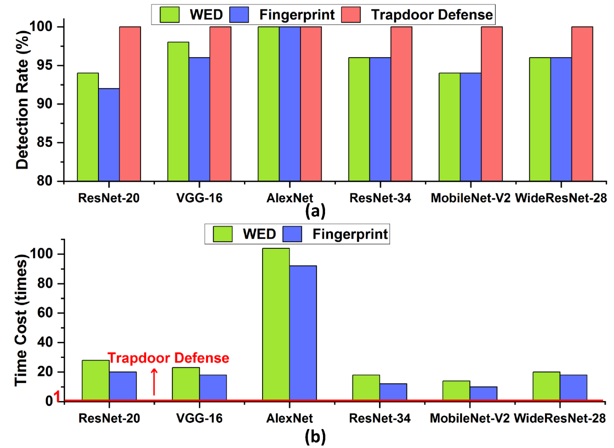

Figure 8: The comparison of detection rate and time cost against existing representative detection approaches.

neurons. We assume the oracle attacker is also fully aware of details about enhanced neurons in addition to honey neurons. *The rationality is that: if the attacker circumvents honey neurons which offer a shortcut for BFA, and enhanced neurons that indicate a much longer path for bit-flips, he or she will be forced to create a path from the (remaining) normal neurons insensitive to BFA. This could lead to the unreasonable amount of effort for attacker under real hardware constraints.*

Specifically, the hybrid countermeasure involves two steps: (1) Selecting honey neurons and normal neurons that need enhancement; and (2) Embedding honey neurons and enhanced neurons into the DNN model. In step 1, we first categorize neural network layers into three levels (high/mid/low) according their sensitivity to attack. A layer's sensitivity can be measured by the sum of the absolute value of its top n weight gradient. The larger it is, the more sensitive a layer is. We empirically choose the number of layers for each level. For high-sensitivity layer, we treat all neurons as honey neurons. For mid-sensitivity layers, we identify the top k_h activated neurons (e.g., $k_h = 10\%$) as honey neurons. In addition, enhanced neurons are selected from the top k_e activated neurons (e.g., $k_e = 50\%$) among the remaining $1 - k_h$ neurons. We leave all neurons in low-sensitivity layers untouched (normal neurons). In step 2, we set a large expanding coefficient ($\gamma > 1$) for honey neurons but a smaller one ($0 < \gamma < 1$) for enhanced neurons. The purpose is to suppress the sensitivity of enhanced neurons that might originally have high sensitivity.

We evaluate it against oracle DeepHammer attack based on the following settings: $\gamma = 2$ ($\gamma = 0.5$) for honey neurons (enhanced neurons), the 1st and 8th layers as the high-sensitivity layers, 2nd, 3rd, 4th layers as the mid-sensitivity layers, and the remaining layers as the low-sensitivity layers. As Table 6 shows, the number of bit-flips needed by oracle attacker increases as the percentage of honey weights increases, for example, $17.6\times$ with merely 2.4% honey weights compared to clean models without trapdoor. We observe the similar trend in other models (See Appendix D). This means that the

Table 6: The overhead of trapdoor and the number of bit-flips under the oracle attack (ResNet-20).

k_h (%)	0 (baseline)	5%	10%	15%	20%
No. of bit-flips	14	82	125	169	243
Num. of honey weights	-	5K (1.8%)	5.5K (2%)	5.9K (2.2%)	6.3K (2.4%)
Num. of honey neurons	-	22 (3.2%)	24 (3.5%)	26 (3.8%)	28 (4%)

defense can significantly raise the bar for the attacker to exert faults injection physically at the cost of more honey neurons.

7.4 Limitations

While our *NeuroPots*-enabled trapdoor defense can effectively trap bit-flips and repair the model online at a low cost, it has a few known limitations. First, the fine-tuning-based honey neuron embedding may incur a retraining cost in large and complex models. Although the retraining-free one-shot method lowers the cost, both methods still lead to marginal accuracy drop. Second, there is still time overhead added into the original inference, e.g. higher overhead in small models than large ones (9.7% for ResNet-20 v.s. 1.1% for MobileNet-V2). Third, while our defense well defeats a series of the state-of-the-art bit-flip attacks which always attempt to tamper the model by flipping a minimum amount of weight bit-flips, including the latest BFA flipping even less bits within only one layer [41], our evaluated adaptive attacks are limited to weight gradient-based bit-flip attacks. Considering the existence of other attacks, e.g. bit-flip trojan attack by hacking model weights and input simultaneously [42] (different threat model), and non-gradient attacks, the *NeuroPots*-based defense idea can be further explored along this direction.

8 Related Work

Deep Neural Networks (DNNs) are vulnerable to both data- and model-centric attacks. While the defenses against the former, such as evasion [8, 9, 10], poisoning [43, 44], backdoor/trojan [11, 12, 13, 45, 46] have been intensively studied, this does not hold for the latter exploiting hardware-based fault injection techniques [14, 15, 16, 17, 47, 48]. Along this direction, the recent BFAs crush quantized DNNs by leveraging greedy gradient-based bit search and precise row-hammer to flip the least number of weight bits stored in DRAMs.

There exist a few attempts to directly address BFA which can be grouped into two categories: fault-tolerance enhancement and fault detection. In the first category, it often requires costly binarization-aware training [49] or BFA-aware weight reconstruction [50], leading to either noticeable accuracy drop or limited robustness improvement. For the second category, it either requires detecting all model weights [38, 40, 51] to ensure detection rate or detecting weights within sensitive layers only [22, 39] through embedding signatures or training machine learning models for detection. They either suffer from high overhead or false negative detection. Moreover, they cannot realize high-accurate real-time model recovery. As evaluated in Section 7.2, *NeuroPots*-based trapdoor defense outperforms them in detection rate and overhead significantly.

Hardware countermeasures can also prevent physical fault injection channels. However, general rowhammer defenses such as Error-Correcting Code and Intel SGX can be broken by new types of rowhammer attacks [52, 53], not to mention the non-trivial hardware modifications like additional/probabilistic refreshes, memory controller redesign, etc.

Honeypot-based proactive defense has been applied in many fields like IoT security [54, 55], cloud security [56, 57], and network security [58, 59]. Recently “honeypot" is also used to detect input-based adversarial attacks in deep learning [60]. However, our work differs in two aspects from theirs: 1) problem– bit flips inside DNN models v.s. pertubations of input data outside DNNs; 2) approach– crafting honey neurons inside DNNs vs. input trigger-based backdoor training.

9 Conclusion

In this work, we propose *NeuroPots*–a proactive defense concept against the emerging bit-flip attacks (BFAs) dedicated to quantized DNNs. Based on *NeuroPots*, we design a trapdoor-enabled defense framework to efficiently detect and mitigate BFAs. Our method embeds a very few well-designed honey neurons as vulnerabilities into DNN models. These vulnerabilities are trapdoors that largely attract attackers to inject bit-flips to the weights associated with honey neurons. Since most bit-flips are captured by trapdoors, defenders can detect faults and recover model accuracy effectively at run time. Extensive experimental results show that our trapdoor-enabled defense achieves high detection (mitigation) rate and extremely low overhead against a variety of static and adaptive attacks.

10 Acknowledgements

We thank our shepherd and the anonymous reviewers for their comments and feedback. This work is supported in part by the National Science Foundation (NSF) under grant number CCF-2011236, CCF-2006748 and CCF-1909854.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, p. 436, 2015.

[2] C. Szegedy, “An overview of deep learning,” *AITP 2016*, 2016.

[3] D. Silver and D. Hassabis, “Alphago: Mastering the ancient game of go with machine learning,” *Research Blog*, vol. 9, 2016.

[4] https://research.fb.com/category/facebook-ai-research-fair/.

[5] *Google TPU*. https://cloud.google.com/tpu.

[6] https://www.microsoft.com/en-us/research/research-area/artificial-intelligence/.

[7] Y. Liu, L. Wei, B. Luo, and Q. Xu, "Fault injection attack on deep neural network," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 131–138, IEEE, 2017.

[8] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *In International Conference on Learning Representations*, 2015.

[9] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pp. 372–387, IEEE, 2016.

[10] F. Tramèr, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "The space of transferable adversarial examples," *arXiv preprint arXiv:1704.03453*, 2017.

[11] T. Gu, B. Dolan-Gavitt, and S. Garg, "Badnets: Identifying vulnerabilities in the machine learning model supply chain," *In NIPS Workshop.*, 2017.

[12] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *arXiv preprint arXiv:1712.05526*, 2017.

[13] Y. Ji, X. Zhang, and T. Wang, "Backdoor attacks against learning systems," in *2017 IEEE Conference on Communications and Network Security (CNS)*, pp. 1–9, IEEE, 2017.

[14] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Practical fault attack on deep neural networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2204–2206, 2018.

[15] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraș, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 497–514, 2019.

[16] F. Yao, A. S. Rakin, and D. Fan, "Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pp. 1463–1480, 2020.

[17] W. Liu, C.-H. Chang, F. Zhang, and X. Lou, "Imperceptible misclassification attack on deeplearning accelerator by glitch injection," in *Proceedings of the 56th Annual Design Automation Conference*, DAC '20, ACM, 2020.

[18] A. S. Rakin, Z. He, and D. Fan, "Bit-flip attack: Crushing neural network with progressive bit search," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1211–1220, 2019.

[19] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

[20] A. S. Rakin, Z. He, J. Li, F. Yao, C. Chakrabarti, and D. Fan, "T-bfa: Targeted bit-flip adversarial weight attack," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[21] M. Ribeiro, K. Grolinger, and M. A. Capretz, "Mlaas: Machine learning as a service," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pp. 896–902, IEEE, 2015.

[22] Q. Liu, W. Wen, and Y. 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*, pp. 1–8, 2020.

[23] L. Batina, S. Bhasin, D. Jap, and S. Picek, "Csi neural network: Using side-channels to recover your artificial neural network information," *Cryptology ePrint Archive*, 2018.

[24] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pp. 2003–2020, 2020.

[25] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pp. 1–9, 2016.

[26] A. Holding, "Arm security technology, building a secure system using trustzone technology," 2009.

[27] *Arm TrustZone Technology*. https://developer.arm.com/ip-products/security-ip/trustzone.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations*, 2015.

[29] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)," *URL http://www. cs. toronto. edu/kriz/cifar. html*, vol. 5, p. 4, 2010.

[30] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[31] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual

learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[33] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[34] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[35] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

[36] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *British Machine Vision Conference 2016*, British Machine Vision Association, 2016.

[37] Y. Li, M. Li, B. Luo, Y. Tian, and Q. Xu, "Deepdyve: Dynamic verification for deep neural networks," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 101–112, 2020.

[38] H. Chen, C. Fu, B. D. Rouhani, J. Zhao, and F. Koushanfar, "Deepattest: an end-to-end attestation framework for deep neural networks," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 487–498, IEEE, 2019.

[39] M. Javaheripi and F. Koushanfar, "Hashtag: Hash signatures for online detection of fault-injection attacks on deep neural networks," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2021.

[40] J. Li, A. S. Rakin, Z. He, D. Fan, and C. Chakrabarti, "Radar: Run-time adversarial weight attack detection and accuracy recovery," in *DATE*, 2021.

[41] J. Bai, B. Wu, Y. Zhang, Y. Li, Z. Li, and S.-T. Xia, "Targeted attack against deep neural networks via flipping limited weight bits," in *International Conference on Learning Representations*, 2021.

[42] H. Chen, C. Fu, J. Zhao, and F. Koushanfar, "Proflip: Targeted trojan attack with progressive bit flips," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 7718–7727, 2021.

[43] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li, "Manipulating machine learning: Poisoning attacks and countermeasures for regression learning," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 19–35, IEEE, 2018.

[44] C. Yang, Q. Wu, H. Li, and Y. Chen, "Generative poisoning attack method against neural networks," *arXiv preprint arXiv:1703.01340*, 2017.

[45] W. Li, J. Yu, X. Ning, P. Wang, Q. Wei, Y. Wang, and H. Yang, "Hu-fu: Hardware and software collaborative attack framework against neural networks," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 482–487, IEEE, 2018.

[46] C. Liao, H. Zhong, A. Squicciarini, S. Zhu, and D. Miller, "Backdoor embedding in convolutional neural network models via invisible perturbation," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2020.

[47] Y. Luo, C. Gongye, Y. Fei, and X. Xu, "Deepstrike: Remotely-guided fault injection attacks on DNN accelerator in cloud-fpga," in *2021 58th ACM/ESDA/IEEE Design Automation Conference (DAC)*.

[48] A. S. Rakin, Y. Luo, X. Xu, and D. Fan, "Deep-dup: An adversarial weight duplication attack framework to crush deep neural network in multi-tenant FPGA," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021.

[49] Z. He, A. S. Rakin, J. Li, C. Chakrabarti, and D. Fan, "Defending and harnessing the bit-flip based adversarial weight attack," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14095–14103, 2020.

[50] J. Li, A. S. Rakin, Y. Xiong, L. Chang, Z. He, D. Fan, and C. Chakrabarti, "Defending bit-flip attack through dnn weight reconstruction," in *57th Design Automation Conference (DAC)*, IEEE, 2020.

[51] F. S. Hosseini, Q. Liu, F. Meng, C. Yang, and W. Wen, "Safeguarding the intelligence of neural networks with built-in light-weight integrity marks (lima)," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 1–12, IEEE, 2021.

[52] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 245–261, IEEE, 2018.

[53] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[54] R. Vishwakarma and A. K. Jain, "A honeypot with machine learning based detection framework for defending iot based botnet ddos attacks," in *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pp. 1019–1024, IEEE, 2019.

[55] X. Luo, Q. Yan, M. Wang, and W. Huang, "Using mtd and sdn-based honeypots to defend ddos attacks in iot," in *2019 Computing, Communications and IoT Applications (ComComAp)*, pp. 392–395, IEEE, 2019.

[56] V. Mahajan and S. K. Peddoju, "Integration of network intrusion detection systems and honeypot networks for cloud security," in *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pp. 829–834, IEEE, 2017.

[57] P. S. Negi, A. Garg, and R. Lal, "Intrusion detection and prevention using honeypot network for cloud security," in *2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pp. 129–132, IEEE, 2020.

[58] J. Bao, C.-p. Ji, and M. Gao, "Research on network security of defense based on honeypot," in *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*, vol. 10, pp. V10–299, IEEE, 2010.

[59] A. Mairh, D. Barik, K. Verma, and D. Jena, "Honeypot in network security: a survey," in *Proceedings of the 2011 international conference on communication, computing & security*, pp. 600–605, 2011.

[60] S. Shan, E. Wenger, B. Wang, B. Li, H. Zheng, and B. Y. Zhao, "Gotta catch'em all: Using honeypots to catch adversarial attacks on neural networks," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 67–83, 2020.

A Number of Bit-flips under BFAs

In Table 8, we list the number of bit-flips generated by three basic BFAs and four adaptive attacks in clean models (w/o trapdoor) and the trapdoored models (w/ trapdoor) on various DNN structures. For basic attacks, we can observe that the number of bit-flips in the trapdoored model is less than the corresponding clean model on average for most DNN structures. For example, in ResNet-20, DeepHammer needs to flip an average of 16 bits to degrade the clean model's accuracy to random guessing, while it needs 5 bits on the trapdoored models. This is because our defense intentionally implants more vulnerable points into the model to attract attackers, which makes attackers achieve their goals more easily. Note different models have different resistance abilities to BFAs, causing varied numbers of needed bit-flips. Since attackers usually cannot know how many bit-flips are needed to achieve their goal in a clean targeted model, it is hard to perceive trapdoors by attackers. For adaptive attacks, we can observe that attackers need to flip more bit-flips to achieve their goal, compared to clean and trapdoored models under basic attacks.

B Detailed Settings of Trapdoor Techniques

Detailed settings of the trapdoor techniques are listed in Table 7. We can observe that the optimal γ based on the guidelines described in Section 6.1.3 is generally (1, 3].

C Results of Defending against Adaptive Attacks with Different Attack Settings

We further evaluate the defense performance of the trapdoor-enabled defense against various adaptive attacks by using three representative DNN models and two datasets–CIFAR-10 and ImageNet.

C.1 Defending against BBA

As Fig. 9 shows, the detection rate of our defense is always 100%. When t = 90%, the mitigation success rate slightly drops, but the number of bit-flips of attack increases $> 12.2\times$ (we set the maximum number of bit-flips is 500 in our experiments), $13.9\times$, $14.6\times$ compared to baseline on VGG-16, AlexNet, ResNet-34, respectively.

C.2 Defending against NBA

Results for R-NBA. In Fig. 10, we can observe that, when $k_b = 30\%$, the mitigation success rate of our defense decreases by $\sim 20\%$, while the number of bit-flips of the attack rises by $> 12.2\times$, $10.1\times$, $13.9\times$ on VGG-16, AlexNet, ResNet-34, respectively.

Results for F-NBA. The results are shown in Fig.11. As Fig. 11 (a) shows, when test times is 100, the attacker only detects a small percentage of honey neurons on three models. Therefore, in Fig 11 (b) and (c), we can observe that the detection rate of our defense is still 100%, and the mitigation success rate only has negligible degradation on three models.

D Defending against Oracle Attack

Table 9 reports the detailed settings and results of hybrid countermeasures against oracle attack to test the bound of our trapdoor defense. As a result, if oracle attacker bypasses our functional neurons (honey and enhanced neurons), the needed bit-flips (or the efforts needed in physical fault injection) of the attacker will increase by $> 12.2\times$, $12.1\times$, $13\times$ on VGG-16, AlexNet, ResNet-34, respectively.

Table 7: Detailed configurations of NeuroPots on different DNN structures and datasets

Dataset	Structure	N	N_l	γ
CIFAR-10	ResNet-20	2	10	3
	VGG-16	10	10	3
ImageNet	AlexNet	25	8 (all layers)	2
	ResNet-34	15	10	3
	MobileNet-V2	2	10	1.1
Google Speech Command	WideResNet-28	10	10	3

Table 8: Number of bit-flips (mean±standard deviation) of clean model (w/o trapdoor) and trapdoor model (w/ trapdoor) under various basic BFAs and adaptive attacks for different DNNs.

Basic Attacks							
Trapdoor	Attack	CIFAR-10			ImageNet		Google Speech Command
		ResNet-20	VGG-16	AlexNet	ResNet-34	MobileNet-V2	WideResNet-28
w/o trapdoor	BFA	13±6	32±11	13±4	9±2	3±1	5±3
	T-BFA	13±5	35±11	13±5	8±2	3±1	5±4
	DeepHammer	16±6	41±13	16±5	11±3	3±1	6±3
w/ trapdoor	BFA	4±1	10±2	10±4	5±1	3±1	3±1
	T-BFA	4±1	11±2	10±4	6±1	3±1	4±2
	DeepHammer	5±1	22±9	13±5	8±1	3±1	3±1
Adaptive Attacks							
w/ trapdoor	BBA	17±7	75±16	31±6	16±5	3±1	8±3
	R-NBA-b	20±8	80±15	29±8	18±4	3±1	11±2
	R-NBA-a	17±6	62±12	17±5	13±3	3±1	8±2
	F-NBA	19±6	60±13	20±6	12±3	3±1	7±2

Table 9: Configuration details and results of our hybrid countermeasures under oracle attack.

Models	High-sensitivity layers	Mid-sensitivity layers	γ for honey/ enhanced neurons	k_h/k_e	Num. of bit-fllips		Honey weights
					w/o trapdoor	w/ trapdoor	w/ trapdoor
VGG-16	[1]	[2,3,4]	5/0.5	20/50	41	>500	51K (0.15%)
AlexNet	[1]	[2,3,4]	2/0.5	30/50	15	181	576K (1.06%)
ResNet-34	[1,10]	[2,3,4,5,8,32]	2/0.5	20/50	10	130	92K (0.42%)

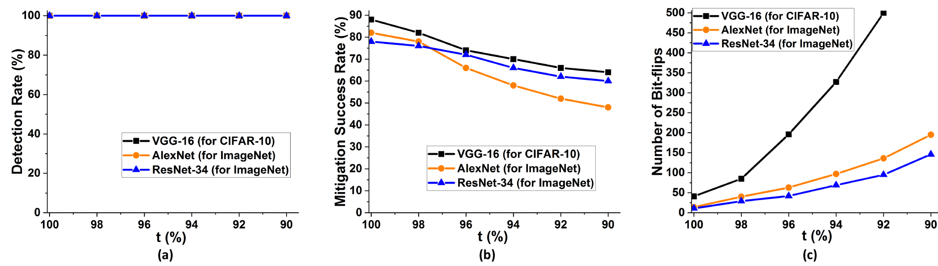

Figure 9: Results of the trapdoor-enabled defense against bit-level bypass (BBA) expert attack on VGG-16, AlexNet, ResNet-34. a) Detection rate of our defense. b) Mitigation success rate of our defense. c) Number of bit-flips of the attack.

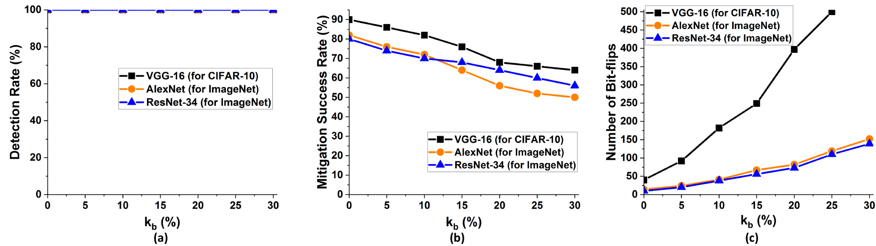

Figure 10: Results of the trapdoor-enabled defense against neuron-level bypass expert attack (R-NBA-b) with bypassing top-ranked neurons on VGG-16, AlexNet, ResNet-34. a) Detection rate of our defense. b) Mitigation success rate of our defense. c) Number of bit-flips of the attack.

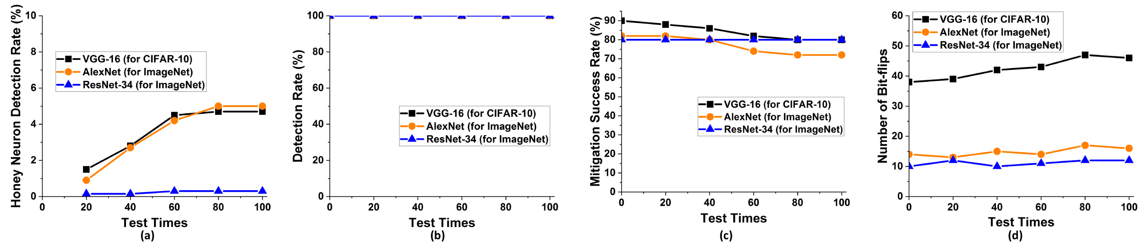

Figure 11: Results of the trapdoor-enabled defense against neuron-level bypass expert attack with bit-flips frequency-based neuron bypass (F-NBA) on VGG-16, AlexNet, ResNet-34. a) Honey neurons detection rate of the attack b) Detection rate of our defense. c) Mitigation success rate of our defense. d) Number of bit-flips of the attack.