ModelShield: A Generic and Portable Framework Extension for Defending Bit-Flip based Adversarial Weight Attacks

Yanan Guo
University of Pittsburgh
Pittsburgh, PA, USA
yag45@pitt.edu

Liang Liu
University of Pittsburgh
Pittsburgh, PA, USA
lil125@pitt.edu

Yueqiang Cheng
NIO
San Jose, CA, USA
strongerwill@gmail.com

Youtao Zhang
University of Pittsburgh
Pittsburgh, PA, USA
zhangyt@cs.pitt.edu

Jun Yang
University of Pittsburgh
Pittsburgh, PA, USA
juy9@pitt.edu

***Abstract*—Bit-flip based adversarial weight attack (BFA) has become one of the most serious threats to Deep Neural Network (DNN) security. By utilizing Rowhammer to flip the bits of DNN weights stored in memory, the attacker can turn a functional DNN into a random output generator. Previous works proposed to retrain or reconstruct weights to make DNNs more robust against BFA, e.g., the attacker may need to flip 10× more weights now to make a DNN malfunction. However, these designs are only weakening BFA instead of preventing it.**

In this work, we propose ModelShield, a new defense mechanism against BFA, based on protecting the integrity of weights using hash verification. If there exists a BFA-free environment, ModelShield verifies the integrity of weights every time when the DNN model is transferred into this environment; if not, ModelShield performs real-time integrity verification. Since integrity verification can slow down a DNN inference by up to 7×, we further propose two optimizations for ModelShield. We implement ModelShield as a lightweight software extension that can be easily installed into popular DNN frameworks. We test both the security and performance of ModelShield, and the results show that it can effectively defend BFA with less than 2% performance overhead.

I. Introduction

Recently, deep neural networks (DNNs) based machine learning (ML) algorithms have shown their great potential in multiple fields, such as object recognition [1], [19] and natural language processing [6], [10], enabling the development of applications in critical domains including finance, biology, and autonomous driving. Instead of investing in their own DNN models, a lot of small companies and personal users prefer to use Machine-Learning-as-a-Service (MLaaS) platforms, where ML applications run on the cloud server and can be accessed by remote users through certain interfaces. MLaaS significantly reduces the effort and cost of developing and maintaining ML applications locally. However, there is usually more than one application running on the cloud server at the same time, and the hardware resources on the server are shared between the ML application and other co-running applications. With this setup, the internal DNN model of an ML application, which is typically stored in the main memory of the server, can be modified by the co-located malicious application (the attacker) indirectly, using Rowhammer [11].

Rowhammer is a security exploit that alters the 1-bit data stored in a memory cell by repeatedly accessing cells in its neighboring rows. Rowhammer allows an user to flip other users' data bits in memory even without write permission to

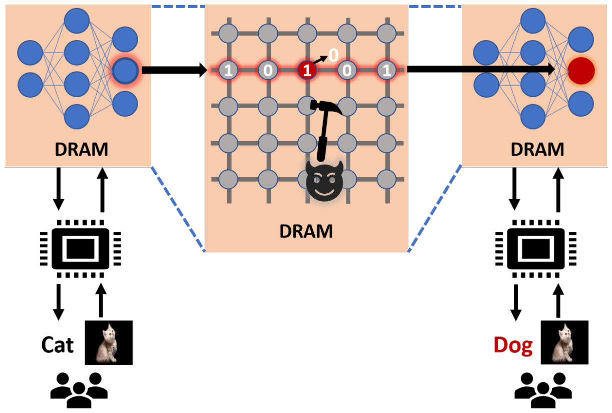

Fig. 1: The result of the attack: after flipping a bit of a weight in memory, the classification output is changed from "cat" to "dog", given a picture of a cat as the input.

them. Previous works have shown that Rowhammer attacks can successfully work on server memory devices [2], [7], [34]. It has also been proved that an attacker can utilize the Rowhammer attack to flip the bits of DNN weights and significantly reduce the inference accuracy or even make the DNN a random output generator [33], as shown in Figure 1.

Original bit-flip based adversarial weight attacks randomly choose weights in a DNN model to flip their bits. Recently, a new Bit-Flip Attack (BFA) [22], [24] was proposed where the attacker efficiently identifies and flips only a small number of *most vulnerable bits* in a DNN model. Several defense mechanisms against BFA have been proposed, including binarization-aware training [9], and quantization based weight reconstruction [14]. However, these methods are only making DNN models more robust against BFA instead of strictly protecting them from BFA. For example, according to [9], without any defense, a BFA attacker only needs to modify 28 weights in a ResNet-20 model to make it malfunction; in contrast, with binarization-aware training, the attacker needs to modify over 500 weights. However, modifying 500 weights out of 305,200 weights in ResNet-20 (0.2%) can be easily realized using Rowhammer attacks, making the protected DNN still vulnerable to BFA. Thus, there is still a lack of effective defense mechanism for BFA. Although we can rely on hardware defenses against all Rowhammer attacks, it usually takes a long time to market any hardware modification, and sometimes upgrading the cloud infrastructures is very expensive, leaving the use of current MLaaS platforms still insecure.

Challenge. One way to prevent BFA is to protect the integrity of DNN weights using hash verification. However, a naive hash verification design could cause both security and performance problems. First, using unkeyed hashes for verification may not protect the model: the attacker could modify the hashes stored in memory as well to ensure that they match the modified weights. Thus, one may consider using a keyed hash and keeping the key on chip instead of storing it in memory, similar to SGX. However, this requires hardware-level modification, which significantly complicates the protection. Second, to avoid the time-of-check to time-of-use problem, we need to perform real-time hash verifications. Unfortunately, this can slow down the inference by up to $7\times$, making it an impractical protection for MLaaS platforms.

In this work, we propose ModelShield, a lightweight defense mechanism that can *strictly* prevent BFA by protecting the integrity of DNN weights. ModelShield is implemented as a software extension that can be simply added into modern frameworks (e.g., Pytorch [21]). We overcome the mentioned security and performance challenges by thoroughly analyzing the strength of BFA and optimizing performance while maintaining security based on the analysis. Specifically, we first prove that a BFA attacker is not able to precisely modify the hash to make it match the corrupted weight data, which gives us a chance to protect DNN weights without a keyed hash and hardware modification. Second, we summarize the critical features of hashes for defending BFA and explore hashes that not only have these features but also good performance. Third, to further reduce the verification overhead, we build a hash tree in software and find the tree structure that provides optimal performance.

Since most DNN inferences are performed on GPUs, we implement ModelShield in a CUDA kernel and build a script to link this kernel with modern frameworks such as Pytorch. With ModelShield, programmers can verify the integrity of weights in one-line python code. We test ModelShield with popular DNN models, and the experimental results show that using ModelShield can successfully protect DNN models from BFA with less than 2% performance overhead and zero accuracy degradation. ***The kernel and the script will be released after this paper is published.***

II. Background and Prior Works

A. Rowhammer Attack

Modern DRAM-based memory chips consist of a two-dimensional array of cells. Each cell stores 1-bit information, represented by the charge of the capacitor in the cell. In 2014, it was found out that current DRAMs are vulnerable to disturbance errors induced by adjacent row activation [11]. More specifically, activating a row in a DRAM bank can cause a little disturbance in its neighboring row; with frequently activating/accessing the same row (i.e. hammering the same row), the disturbance on the neighboring row will accumulate and eventually become significant enough to flip the stored bits in this row, before it gets refreshed. With Rowhammer, attackers are able to change the memory data of a co-located application without even accessing it directly.

Rowhammer has already been successfully utilized to demonstrate many attacks [2], [33], [34]. It has been proved that Rowhammer attacks can work on both DDR3 and DDR4 memory, and even on Error-correcting code memory (ECC memory) [3]. In addition, Fan et al. discovered that Rowhammer attacks can be used to modify the weights of a functional DNN and make it a random output generator [33].

B. Bit-Flip based Adversarial Weight Attack

Adversarial weight attack is one of the main challenges on DNN security: even small changes to weights can lead to significant differences in inference accuracy [22]. The bit-flip based adversarial weight attack (bit-flip attack for short) is one of such adversarial attacks. This attack performs weight modification by flipping the bits of DNN weights stored in memory, utilizing well-developed Rowhammer tools.

The original bit-flip attack randomly selects weights in a DNN as the target of error injection. However, this method only works efficiently on floating-point DNN models. Quantized models are very robust against it due to their fixed precision: one bit-flip on the most significant exponent bit of a random weight in a floating-point ResNet-18 can make it totally malfunction on ImageNet; however, 100 bit-flips on a quantized ResNet-18 only causes 0.6% accuracy degradation. Thus, Adnan et al. proposed a more efficient Bit-Flip Attack [22] (aka. BFA), which focuses on compromising a quantized DNN model with only several bit-flips.

Given an N_q-bit quantized DNN model with L convolutional/linear layers, the main goal of BFA is to find the optimal combination of weight bits in it to perform the attack and thus maximize the inference loss of the perturbed DNN model. This can be represented as an optimization problem:

$$\begin{aligned} \max_{\{\hat{\mathbf{B}}_l\}} \; & \mathcal{L}\left(f(\mathbf{x}; \{\hat{\mathbf{B}}_l\}_{l=1}^{L}\right), \mathbf{t}) - \mathcal{L}\left(f(\mathbf{x}; \{\mathbf{B}_l\}_{l=1}^{L}), \mathbf{t}\right) \\ \text{s.t.} \quad & \sum_{l=1}^{L} \mathcal{D}(\hat{\mathbf{B}}_l, \mathbf{B}_l) \leq N_\epsilon \end{aligned} \tag{1}$$

where $\mathbf{B}$ is the bit-wise representation of the quantized weights $\mathbf{W}$, and $\hat{\mathbf{B}}$ is for the perturbed weights. $\mathbf{x}$ and $\mathbf{t}$ are the input and the target output vectors, respectively. Given $\mathbf{x}$ as the input, $f(\mathbf{x}; \{\hat{\mathbf{B}}_l\}_{l=1}^{L})$ represents the inference output using the DNN model with perturbed weights $\{\hat{\mathbf{B}}_l\}_{l=1}^{L}$. Additionally, $\mathcal{L}(\cdot,\cdot)$ computes the loss between the inference output and the target output. $\mathcal{D}(\cdot,\cdot)$ calculates the Hamming distance between two binary tensors, and $N_\epsilon \in \mathbb{N}$ is the maximum Hamming distance allowed in this DNN model.

BFA solves this problem using Progressive Bit Search (PBS) which combines the gradient ranking and progressive search to find the most vulnerable bits. PBS is a two-step algorithm consisting of 1) in-layer search, and 2) cross-layer search. In each attack iteration, PBS first searches in a certain layer to identify bits with highest gradients $(\text{argmax}_{\mathbf{B}_l} |\nabla_{\mathbf{B}_l} \mathcal{L}|)$ as vulnerable bits. This search is performed on each layer independently. Then, PBS compares the selected vulnerable bits from each layer and identifies the most vulnerable bits across all layers by directly checking the loss increment.

C. Previous Defenses

Binarization-aware training: Zhezhi et al. made an important observation that BFA is prone to identify vulnerable bits in close-to-zero weights, and modify them to be large values [9]. Based on this observation, they proposed to use binarization-aware training to defend against BFA. In this training method, weights that are originally in floating point are converted to be represented by a binary-based format; the method can be mathematically described as:

$$\begin{aligned} \text{Forward: } & w_{l,i}^{\text{b}} = \mathbb{E}(|\mathbf{W}_l^{\text{fp}}|) \cdot \text{sgn}(w_{l,i}^{\text{fp}}) \\ \text{Backward: } & \frac{\partial \mathcal{L}}{\partial w_{l,i}^{\text{b}}} = \frac{\partial \mathcal{L}}{\partial w_{l,i}^{\text{fp}}} \end{aligned} \tag{2}$$

in which $w_{l,i}^{\text{fp}}$ and $w_{l,i}^{\text{b}}$ denote the floating-point and binarized weight, respectively, and sgn() is the sign function. With this method, weights in each layer of the model can only be $\pm\mathbb{E}(|\mathbf{W}_l^{\text{fp}}|)$, i.e. weights are *far from zero*. Thus, the model becomes more robust against BFA. Note that since using binarization-aware training can significantly decrease the inference accuracy, the authors also proposed a relaxation to the weight binarization which can help improve accuracy.

Weight reconstruction: As mentioned, when the attacker flips a bit in a weight, it induces a change of Δw on the target weight, which will affect the loss. Thus, to defend BFA, Jingtao et al. proposed weight reconstruction method which can reduce the Δw caused by a bit-flip and thus the overall increase in loss [14]. This weight reconstruction consists of three steps: 1) averaging, which can spread the effect of $|\Delta w|$ on a group of size G so that it only causes a small change of $|\Delta w/G|$ on the mean; 2) quantization, which can cancel the effect of this $|\Delta w/G|$ change on the quantized mean; 3) clipping, which restricts all the weights to a small range around the quantized mean.

Limitations of previous works: The discussed previous defenses focus on modifying DNN models themselves to make them more robust against BFA. However, this kind of method can only weaken BFA, rather than preventing it. As an example, without any defense mechanism, the BFA attacker only needs to flip the most significant bits (MSBs) of about 28 weights in a ResNet-20 model to make it generate random outputs on CIFAR-10. In contrast, with binarization-aware training, the attacker now needs to flip at least 500 MSBs. This defense mechanism significantly improves DNN security; however, there are over 1MB weights in ResNet-20. Although previous work reported that not all memory cells are vulnerable to Rowhammer, it is still not difficult for the attacker to find 500 weights with vulnerable MSBs, which is less than 0.2% of the total weights. From this perspective, previous defenses make weak assumptions on the attacker's strength and cannot strictly protect DNN models from BFA.

Note that defense methods targeting general fault-injection attacks on DNN models can also be used to defend BFA, such as [15]. However, they always come with high overhead.

III. The Design of ModelShield

A. Threat Model

We assume a white-box attacker similar with the ones in previous works [9], [14]. The attacker's goal is to make the co-located DNN application malfunction (i.e. provide wrong inference results) by performing BFA. To achieve this, the attacker should have some knowledge of 1) the architecture, weights, and other parameters of the target DNN model, and 2) the memory mapping mechanism of the MLaaS platform. Thus, the attacker knows which weights of the DNN model should be modified and how to modify them in memory via Rowhammer. In addition, for a DNN application, the DNN model is usually first loaded into CPU memory during initialization, and later transferred into GPU memory to perform inference. Therefore, we use two threat models based on whether GPU memory is vulnerable to Rowhammer attacks. In the *current threat model*, we assume that only CPU memory is vulnerable to Rowhammer attacks and GPU memory is safe, since Rowhammer attacks on GPU have not been discovered yet. However, in case those attacks will be built in the future, we also use a *future threat model* in which we assume both CPU memory and GPU memory are vulnerable. We propose two versions of ModelShield under these two threat models. Detailed justification of these threat models are in Section III-C and Section III-D.

In both threat models, we exclude Rowhammer attacks that focus on attacking the code structure of DNN applications to change the control flow or even cause a Denial-of-Service (DOS) attack, since they are easier to detect and can be defended by previous works [27].

B. Design Overview

Instead of rebuilding the DNN model to make it more robust against BFA, in this work, we aim to propose a new method that can detect and prevent any unauthorized modifications on the weights of a DNN model. To *practically* protect DNN models, this method should meet the following requirements:

1) **Easy implementation.** Rowhammer attacks can be eliminated by hardware defenses. However, it is a long process to test and verify hardware modifications before applying them to commercial processors. In fact, recent memory devices are even more vulnerable to Rowhammer attacks than earlier devices [29]. Thus, if we aim to build a defense method that is available for DNN customers immediately, this design should not include any hardware-level modification, i.e. it should be a simple software-level design.
2) **Compatible with current frameworks.** Most DNN models are trained and used in popular frameworks such as Pytorch [21]. Thus, it is necessary to guarantee that with this new software-level defense method, users are still able to train/use their model in these frameworks.

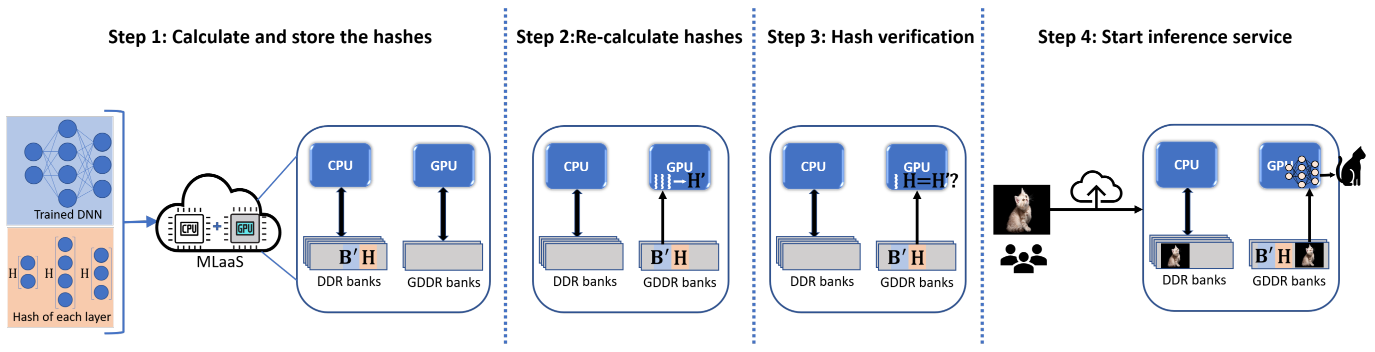
Step 1: Calculate and store the hashes
Step 2:Re-calculate hashes
Step 3: Hash verification
Step 4: Start inference service
Trained DNN
Hash of each layer
MLaaS
CPU
GPU
DDR banks
GDDR banks
B′ H
H=H′?

Fig. 2: Pre-inference integrity verification mechanism; $\mathbf{B}'$ means the weights stored in memory (potentially have been modified by BFA), $\mathbf{H}$ means the stored hash values, and $\mathbf{H}'$ means the re-calculated hash values over $\mathbf{B}'$. DDR and GDDR are the memory devices for CPU and GPU, respectively.

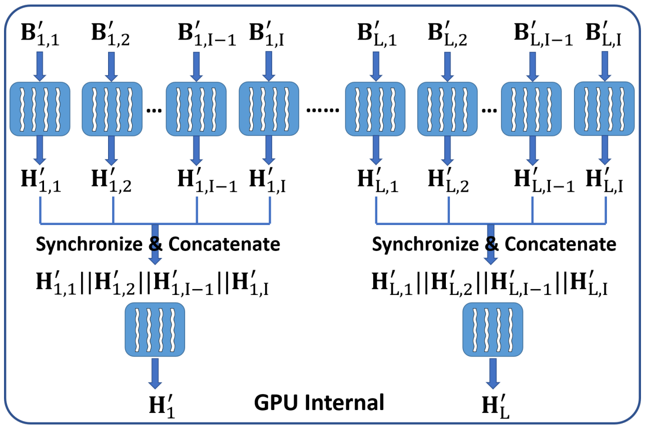

Fig. 3: Parallelizing hash verification in GPU by building a 2-level hash tree; each wavy line represents a GPU thread.

3) **Negligible performance effect.** The inference performance directly affects the competitiveness of an MLaaS platform. Thus, the proposed method should not introduce a significant increase in the inference latency.

Based on these requirements, we propose ModelShield which can defend BFA by verifying the integrity of weights. ModelShield is a software extension that can be easily installed into current frameworks such as Pytorch, and it is designed to only have very little performance overhead (1%) on DNN training/inference. In this section we explain how ModelShield works under each threat model.

Note that in this section we ***only consider the most common cases in which DNN inference is executed on GPU***. Cases in which the inference is finished on CPU or specialized DNN accelerator will be discussed in Section V.

C. ModelShield under The Current Threat Model

As mentioned, our goal is to detect and prevent any weight modifications caused by BFA, i.e. the integrity of weights needs to be protected. Thus, we propose to ***verify the integrity of weights every time after the DNN is transferred into a BFA-free environment (pre-inference verification).***

Due to their high calculation bandwidth, GPUs are widely used for DNN inference. On an MLaaS platform, the service host usually transfers the DNN model from CPU memory to GPU memory before starting the inference service. Once the inference starts, users can send their inputs to the platform and the host will transfer the inputs also to GPU memory to perform the inference on GPU. Fortunately, we have not seen a successful Rowhammer attack targeting GPU memory, and previous BFAs are all done in CPU memory [22], [33]. This is because Rowhammer attacks require the attacker application to be co-running with the victim application on the same computing device. This is naturally supported on CPU, with the help of operating system. However, due to the lack of operating system on GPU, it is very difficult (although not impossible) to run two applications (aka. GPU kernels) simultaneously on GPU [30]. In addition, in contrast to the case on CPU, it is much harder to manipulate the cache on GPU which is necessary for Rowhammer attacks. Therefore, we consider GPU as a *BFA-free environment*. Thus, as long as the DNN model's integrity is verified every time when the model enters this environment, we do not need to worry about BFA while the model is kept in this environment.

A cryptographic hash function $H()$ is an algorithm that takes an input data with any size, and produces a fixed-size output, called the hash value or the digest (e.g., SHA256 [18] is a popular cryptographic hash that generates a 256-bit hash value). Cryptographic hashes (hashes for short) are widely used for protecting data integrity. Thus, here we use them to defend against BFA. The defense algorithm (Figure 2) is explained as follows:

Step 1: After a DNN model is trained, for each layer of the model, calculate one hash value H_l over the weights in this layer ($H_l = H(\mathbf{B}_l)$), and store all the hash values $\{H_l\}_{l=1}^{L}$ together with the weights $\{\mathbf{B}_l\}_{l=1}^{L}$, as part of the model. Then when the model is deployed to an MLaaS platform, these hashes will also be deployed;

Step 2: During the initialization of inference service, after the model is moved into the GPU memory, re-calculate the hashes in GPU over the weights stored in GPU memory, i.e. calculate $H_l' = H(\mathbf{B}_l')$ for each layer;

Step 3: Compare $\{H_l'\}_{l=1}^{L}$ with $\{H_l\}_{l=1}^{L}$ to verify the integrity of weights: if they are equal then $\mathbf{B} = \mathbf{B}'$, no BFA occurred before this DNN model is transferred into GPU memory; if they are different then $\mathbf{B} \neq \mathbf{B}'$, this DNN model is compromised and we need to reload the correct model from the disk storage and restart from Step 2;

Step 4: Start the inference service on this platform. During this service, if the model is never moved out of GPU, we do not need to verify the integrity of weights again[1].

One natural question is that *since the hash values are also stored in memory, why can't a "smart" BFA attacker modify the hash values together with weights to ensure that they still match after the attack*. Theoretically, this is a security concern for ModelShield; however, we argue that this is not feasible in reality due to the following properties of cryptographic hashes [25]:

①**Diffusion**: hash values are guaranteed to be diffused, i.e. even modifying one bit in the input can cause many bits to change in the hash value;

② **Randomness:** each hash value is expected to be random, i.e. there are about same amount of "0" and "1" in it.

Due to these two properties, if the attacker wants to modify the hash value to make it match the modified weights, he needs to flip many bits of the hash value including both $1 \rightarrow 0$ and $0 \rightarrow 1$ flips. However, according to previous works [11], [32], all the memory cells in the same row can only be flipped in one certain direction (either $1 \rightarrow 0$ or $0 \rightarrow 1$). As a hash value is typically smaller than one cache line, its bits are all stored in one memory row. Thus, it is almost impossible for the attacker to get the correct hash value by only performing one-direction bit-flips to all the bits (e.g., for SHA256, the attacker can succeed with the probability of $1/2^{127}$).

We implement this DNN integrity verification method in a GPU kernel and build a script to link this kernel into frameworks such as Pytorch. With this kernel, the integrity of a DNN model can be easily verified with **one-line python code** (for Requirement 1 & 2). Additionally, this verification only needs to be done once (when the model is transferred into GPU memory), then the model will be used to serve many inferences. Thus, the overhead on inference latency is negligible (for Requirement 3).

D. ModelShield under The Future Threat Model

The pre-inference verification design in Section III-C assumes that GPU memory is BFA-free since we have not seen Rowhammer attacks on GPU yet. However, as we mentioned, it is not impossible to run two GPU kernels simultaneously [17], [30]. Although this co-running is not yet stable enough for building Rowhammer attacks on GPU, maybe it will be in the future. Thus, GPU memory may become vulnerable to BFA in the future. In that case, our design will not be effective anymore because after the model's integrity is verified, the model can still be attacked by BFA in GPU memory during the DNN inference, introducing a time-of-check to time-of-use problem [31]. In this section we modify ModelShield to further prevent the future BFA in GPU memory, based on some important observations.

Observation 1 *Rowhammer attacks can only flip the value of a memory cell in one certain direction (1 $\rightarrow$ 0 or 0 $\rightarrow$ 1).*

The root cause of Rowhammer is that repeatedly accessing one row can cause the DRAM cells in the neighboring row to leak charge. In a DRAM chip, there can be two types of cells, including *true-cells* and *anti-cells* [32], depending on the relationship between the voltage of the cell and the represented logic values: a true-cell stores a logic value of 1 as the charged state, and 0 as the discharged state. Thus, charge leaking caused by Rowhammer can only introduce $1 \rightarrow 0$ bit-flips. In contrast, in anti-cells this relationship is in the opposite direction, i.e. Rowhammer can only introduce $0 \rightarrow 1$ bit-flips. As mentioned in Section III-C, cells in the same row are always the same type.

According to this observation, once a BFA attacker modifies a bit of a weight during the inference, he will not be able to flip this bit back to pretend that it was never changed. Thus, if GPU memory is not secure anymore, we can use *post-inference integrity verification* to protect weights: we can verify the integrity of weights every time after the inference is done but before the inference result is sent to users; as long as at this time the verification succeeds, it indicates that no BFA happened during the inference.

This post-inference verification design is secure even when GPU memory is corrupted. However, this design can introduce significant overhead: if users are performing inference of one input at a time, then we need to do a hash verification after the inference of each single input. This can increase the total execution latency by several times (as shown in Section IV). To solve this problem, we further propose two optimizations.

Observation 2 *Non-cryptographic hash functions can also be used for defending BFA.*

In contrast to non-cryptographic hash functions, cryptographic hash functions are able to strictly protect data integrity, and thus can definitely be used to prevent BFA. In cryptography, strict integrity protection requires/indicates that the generated hash value has the property of diffusion and randomness. As explained in Section III-C, these properties are needed for defending BFA. However, integrity protection also indicates the guarantee of collision resistance, i.e. it is computationally infeasible to find two inputs that have the same hash value. However, this property is not necessary for defending BFA, due to the limited ability of Rowhammer attacks: even if the attacker finds two inputs that collide, he is not able to modify the weights accurately to have the colliding values as those two values are likely to be very different.

Optimization 1 *Use high-performance non-cryptographic hash functions to defend BFA.*

As mentioned, we do not necessarily need a cryptographic hash function for defending BFA. We instead only need a hash function with the property of diffusion and randomness.

[1]This is true in most cases, in rare cases where the DNN model cannot fit in GPU memory, the hash needs to be verified every time when a DNN layer is transferred into GPU memory.

This requirement can actually be satisfied by some non-cryptographic hash functions such as xxHash [4]. Most cryptographic hash algorithms are much more complicated than non-cryptographic hashes, and thus have worse performance. For example, Merkle–Damgård construction based cryptographic hashes are known to have extremely long calculation latency [5]. Thus, we instead use non-cryptographic hashes that are highly optimized, such as xxHash. In Section IV, we will show that using xxHash introduces much less overhead compared to using SHA256.

Optimization 2 *Build a hash tree to fully utilize the calculation resource in GPU.*

The nature of hash function is to *compress* an arbitrary-size input data to a fixed-size output. A longer input usually renders longer hash calculation latency, as it requires more iterations of compression (Figure 4). For example, with defining the output length of the hash function as a block, the hash calculation latency with a 3-block input can be about two times of the latency with a 2-block input. This is because the former requires two iterations of compression and the latter only needs one. GPU consists of many computation threads that can work simultaneously. Thus, the hash values of different DNN layers can be generated in parallel, since they are independent. However, the calculation of a certain hash value is not parallelizable because compressions have to be done sequentially; if a layer in the DNN model is very large, it can result in long total hash calculation latency.

Thus, to reduce the hash calculation overhead, we build a small hash tree (for each DNN layer) which can fully utilize the calculation ability of GPU threads to parallelize the hash calculation: for all the weights in a layer ($\mathbf{B}'_l$), we first divide them into several chunks ($\{\mathbf{B}'_{l,i}\}_{i=1}^{I}$) ①; then we calculate the hash of each chunk ($H'_{l,i}$) simultaneously using different threads in GPU ②; after this is done, we concatenate the outputs in ② and get $H'_{l,1}||H'_{l,2}||...||H'_{l,I}$. Then we calculate the hash of this concatenated result as the final hash of this layer (H'_l) ③, as shown in Figure 3. Note that the stored hash values $\{H_l\}_{l=1}^{L}$ (generated before the DNN model is deployed) need to be calculated the same way.

Note that we cannot use too many chunks in ①, i.e. I cannot be too large a number. In the extreme case, if each chunk is 1-block long, there will be no compression in ②, and after ② the output size is same with the input. Then in ③ we still need to do a long calculation/compression. Thus, the problem is how many chunks should we have in ① to reach the best performance, i.e. for an n-block input, if the latency of hashing n-block to 1-block is about $(n-1)\cdot t$, what is the chunk number c that makes the total latency shown as follows the minimum:

$$L_{hash} = \underbrace{(n/c-1)\cdot t}_{L_{hash1}} + \underbrace{(c-1)\cdot t}_{L_{hash2}} \tag{3}$$

Note that L_{hash} is the total latency, and L_{hash1} and L_{hash2} are the hash latencies in ② and ③, respectively. By solving this math problem, we know that when $c = \sqrt{n}$, L_{hash} reaches the minimum. Thus, for each layer of weights, we divide them into $\sqrt{n}$ chunks first, and use a 2-level hash tree to reduce the calculation latency[2].

E. The Protection of ModelShield

A natural concern on ModelShield is that the attacker might be able to modify the control flow of the hash verification code via Rowhammer to modify or even bypass the verification process. Although protecting code integrity is out of our threat model, we argue that the implementation of ModelShield makes it very difficult (if not impossible) for the attacker to achieve this.

First, the hash verification is implemented as a standalone GPU kernel. There is no control flow to decide whether or not to verify; as long as ModelShield is used, the compiled kernel will be linked into the DNN application's address space, and the hash verification calculation will be executed.

Second, the hash calculation does not have control flow. Although at the end of the calculation, we must check if the calculated hash is equal to the expected hash, this can be done in a branchless manner using general compiler techniques to eliminate branching. An example implementation is as follows:

Step 1: XOR the calculated hash with the expected hash, store in Register S.
Step 2: OR all the bits in Register S together (this can be done branchlessly in log2(hash size)), store in Register T.
Step 3: Use Register T to raise a cuda exception (e.g., assert(NOT(T)). This throws a cudaError if the input is 0 (i.e. the hash is mismatched), and terminates the DNN application. After this, the system manager will relaunch this application to reload the model and weights from disk.

IV. Evaluation

In this section, we analyze the security and performance of ModelShield. In the experiments, we use two baselines including 1) the $insecure\_baseline$, in which there is no protection against BFA, and 2) the discussed binarization_aware training mechanism (BAT). We choose BAT because it is more effective than weight reconstruction. We implement BAT using the public BAT source code in [23].

A. Experiment Setup

Datasets: We focus on the most widely used visual datasets CIFAR-10 [12] and ImageNet [13]. CIFAR-10 contains 60K 32×32 RGB images that are evenly distributed in 10 classes. In the experiments, we use 50K images for training the DNN model, and the remaining 10K for testing it. ImageNet contains 1.2M training images divided into 1000 distinct classes.

Networks: We use CIFAR-10 to test VGG-11 [28] and ResNet-20 [8], and use ImageNet to test MobileNetV2 [26]. VGG-11 is a deep sequential model; ResNet-20 and MobileNetV2 are non-sequential models where some layers take

[2]The performance can be further optimized using a deeper hash tree; however, from the experimental results we found that a 2-level tree is good enough.

TABLE I: The inference latency (given one input image), and the verification latency when using different hash functions and setups; each (%) represents the corresponding overhead on inference latency.

	Max # of weights in a layer	Inference latency (ms)	Verification latency (ms) SHA256	Verification latency (ms) xxHash	Verification latency (ms) xxHash+Tree
ResNet-20	36,864	17.10	28.72 (167.95%)	0.92 (5.38%)	**0.02 (0.12%)**
MobileNet	1,280,000	19.51	74.29 (380.08%)	12.52 (64.17%)	**0.19 (0.10%)**
VGG-11	2,359,296	21.03	164.20 (780.79%)	22.12 (105.18%)	**0.37 (1.76%)**

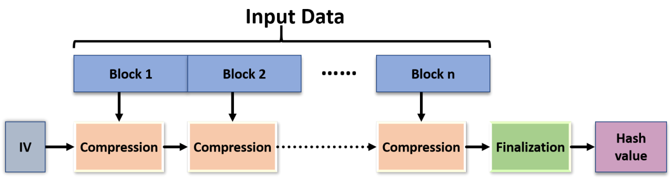

Fig. 4: The internal architecture of Merkle–Damgård construction based cryptographic hashes.

multiple inputs from other layers. We use 8-bit quantization-aware training in all the experiments.

BFA configuration: We test the security of previous works and ModelShield against BFA, using the BFA public codebase [23] (commit: 6ad210c). We use the same attack setup with the one in [22]. For each DNN model, we randomly select 256 input images from the validation set, and use those images to perform BFA and collect the decline of accuracy.

Hardware platform: All the experiments are conducted using Pytorch, running on the platform with an AMD Ryzen 3900XT CPU and an NVIDIA 1080TI GPU.

B. ModelShield Implementation

Hash functions: We evaluate ModelShield with two hash functions, including 1) SHA256 [18], one of the most widely-used cryptographic hash functions with a 256-bit output hash value; 2) xxHash, a high-speed non-cryptographic hash function with excellent diffusion and randomness [4]. We use the 64-bit version of xxHash which gives a 64-bit hash value.

Software implementation: We implement ModelShield in a stand-alone GPU kernel, using CUDA toolkit 10.2 [20]. This toolkit is supported by most frameworks (e.g., Pytorch, TensorFlow). We also write a script to link this CUDA kernel to those frameworks. With this script, users do not need to re-compile the whole framework to support ModelShield. Instead, they can just run the provided script to install ModelShield as an extension to their local framework.

C. Security Analysis

We run BFA to test the security of ModelShield, BAT, and the insecure baseline. As shown in Figure 5, in the insecure baseline without any defense against BFA, 50 bit-flips are enough to compromise a DNN model and convert it into a random output generator (when the accuracy is no more than 10% for CIFAR-10, and 0.1% for ImageNet).

BAT makes a major breakthrough on defending BFA: comparing with the insecure baseline, using BAT makes DNN models much more robust against BFA; less than 20 bit-flips can barely decrease accuracy. However, it is also shown in

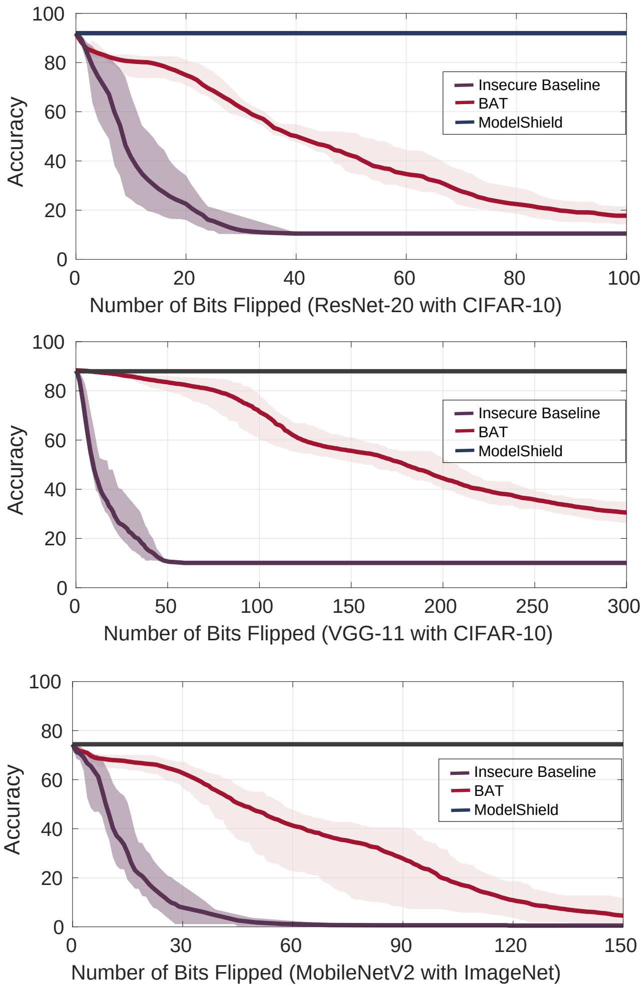

Fig. 5: The BFA result on ResNet-20, VGG-11, and MobileNetV2 with different defense mechanisms. Regions in shadow indicate the error band w.r.t 10 trials.

Figure 5 that, when the number of bit-flips is higher than a certain threshold (about 20 for ResNet-20, 60 for VGG-11, and 30 for MobileNetV2), the accuracy starts to drop very fast with the increase in the number of bit-flips, making the DNN still vulnerable to BFA. Note that our results are slightly different than the results reported in [9].

In contrast, ModelShield is able to detect any bit-flip, and reload the correct model parameters if necessary. Thus, the weights used during the inference are guaranteed to be the unmodified weights, and BFA does not decrease accuracy when using ModelShield.

D. Performance Analysis

Although BAT has relatively weak security protection on DNN models, it does not increase the inference latency. In contrast, ModelShield (under the future threat model) may cause some inference overhead. In this section, we show that this overhead is in fact negligible.

Performance overhead on ResNet-20: ResNet-20 is a relatively small model. As shown in Table I, given one input image, the inference latency on GPU is about 17.10 ms. Using SHA256 to verify the integrity of all the weights will take about 28.72 ms: this is not a significant overhead for pre-inference verification mechanism as it only happens once when the model is transferred into GPU memory. However, for post-inference verification, this latency is added on each inference, which can introduce 167.95% overhead. Instead, using xxHash can reduce this overhead to 5.8%, and building a 2-level hash tree can further reduce it to only 0.12%.

Performance overhead on MobileNetV2 and VGG-11: Comparing with ResNet-20, MobileNetV2 and VGG-11 are much larger models. As mentioned, the size of the largest layer in a model decides the total latency of hash verifications. From Table I, the largest layer of MobileNetV2 and VGG-11 contains 35× and 65× as many weights as the largest layer of ResNet-20, respectively. Thus, hash verifications can generate significant overhead in post-inference verification mechanism: using SHA256 can cause 380.08% and 780.79% overhead on MobileNetV2 and VGG-11, respectively. Even when using xxHash, the overhead can still be 64.17% and 105.18%. These large overheads come from sequentially compressing the weights many times. Therefore, using a hash tree to parallelize this process can significantly reduce the overhead, to only 0.10% and 1.76%, which can be considered negligible.

V. Discussion

Security of DNN inference on CPU/accelerator: Some simple DNN inferences can be directly executed on CPU. In this situation, mature CPU memory integrity protection technologies such as Intel SGX [16] can be used to protect DNN weights. When running DNN inferences on CPUs without the support of such technologies or on DNN accelerators, our post-inference verification mechanism can be slightly modified and used on these platforms.

The effect of implicit memory swap: The runtimes of current DNN frameworks do not implicitly transfer data between CPU and GPU memory. Therefore, ModelShield can be implemented in a standalone GPU kernel. In the future if frameworks start to handle data movement, we will implement ModelShield inside the back-end of each framework to guarantee that data transfer always comes with integrity verification.

The effect of using multiple GPUs: Server platforms usually have multiple GPUs available for users. If a DNN model is split into pieces and transferred to multiple GPUs for inference, we can modify ModelShield to verify the integrity of the corresponding piece on each GPU independently.

VI. Conclusion

In this work, we proposed to use hashes to protect the integrity of DNN weights, and thus defend BFA. We implemented this method in a software extension named ModelShield, and explained the details of ModelShield under the current threat model and under the future threat model in which BFA becomes more powerful. We also designed two optimizations to ensure that ModelShield does not generate unfeasible inference overhead. Finally, our experimental results show that ModelShield can effectively protect DNN models from BFA as well as maintain low inference latency.

References

[1] P. Anderson *et al.*, "Bottom-up and top-down attention for image captioning and visual question answering," in *CVPR*, 2018, pp. 6077–6086.

[2] Y. Cheng *et al.*, "Cattmew: Defeating software-only physical kernel isolation," *TDSC*, pp. 1–1, 2019.

[3] L. Cojocar *et al.*, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *S&P*, 2019, pp. 55–71.

[4] Y. Collet, https://code.google.com/p/xxhash/, 2014.

[5] I. Damgård, "A design principle for hash functions," in *CRYPTO*, 1989, pp. 416–427.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[7] P. Frigo *et al.*, "Trrespass: Exploiting the many sides of target row refresh," in *S&P*, 2020, pp. 747–762.

[8] K. He *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[9] Z. He *et al.*, "Defending and harnessing the bit-flip based adversarial weight attack," in *CVPR*, 2020, pp. 14 095–14 103.

[10] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[11] Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ISCA*, 2014, p. 361—372.

[12] A. Krizhevsky *et al.*, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/ kriz/cifar.html

[13] ——, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84—90, 2017.

[14] J. Li *et al.*, "Defending bit-flip attack through dnn weight reconstruction," in *DAC*, 2020, pp. 1–6.

[15] Y. Li *et al.*, "Deepdyve: Dynamic verification for deep neural networks," in *CCS*, 2020, p. 101–112.

[16] F. McKeen *et al.*, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *HASP*, 2016.

[17] H. Naghibijouybari *et al.*, "Rendered insecure: Gpu side channel attacks are practical," in *CCS*, 2018, pp. 2139–2153.

[18] NIST, *Secure Hash Standard - SHS: Federal Information Processing Standards Publication 180-4*, 2012.

[19] W. Norcliffe-Brown *et al.*, "Learning conditioned graph structures for interpretable visual question answering," in *NeurIPS*, 2018, pp. 8334–8343.

[20] NVIDIA, "Cuda, release: 10.2.89," https://developer.nvidia.com/cuda-toolkit, 2020.

[21] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019, pp. 8024–8035.

[22] A. S. Rakin *et al.*, "Bit-flip attack: Crushing neural network with progressive bit search," in *ICCV*, 2019, pp. 1211–1220.

[23] ——, "Bit-flips attack and defense," https://github.com/elliothe/BFA, 2020.

[24] ——, "T-bfa: Targeted bit-flip adversarial weight attack," *arXiv preprint arXiv:2007.12336*, 2020.

[25] P. Rogaway *et al.*, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," in *FSE*, 2004, pp. 371–388.

[26] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018, pp. 4510–4520.

[27] R. Schilling *et al.*, “Securing conditional branches in the presence of fault attacks,” in *DATE*, 2018, pp. 1586–1591.

[28] K. Simonyan *et al.*, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[29] M. Wang *et al.*, “Dramdig: A knowledge-assisted tool to uncover dram address mapping,” in *DAC*, 2020.

[30] Z. Wang *et al.*, “Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing,” in *HPCA*, 2016, pp. 358–369.

[31] J. Wei *et al.*, “Tocttou vulnerabilities in unix-style file systems: An anatomical study,” in *FAST*, 2005, p. 12.

[32] X.-C. Wu *et al.*, “Protecting page tables from rowhammer attacks using monotonic pointers in dram true-cells,” in *ASPLOS*, 2019.

[33] F. Yao *et al.*, “Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips,” in *USENIX Security*, 2020.

[34] Z. Zhang *et al.*, “Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses,” in *MICRO*, 2020.