AccHashtag: Accelerated Hashing for Detecting Fault-Injection Attacks on Embedded Neural Networks

MOJAN JAVAHERIPI, JUNG-WOO CHANG, and FARINAZ KOUSHANFAR, University of California San Diego, USA

We propose AccHashtag, the first framework for high-accuracy detection of fault-injection attacks on Deep Neural Networks (DNNs) with provable bounds on detection performance. Recent literature in fault-injection attacks shows the severe DNN accuracy degradation caused by bit flips. In this scenario, the attacker changes a few DNN weight bits during execution by injecting faults to the dynamic random-access memory (DRAM). To detect bit flips, AccHashtag extracts a unique signature from the benign DNN prior to deployment. The signature is used to validate the model's integrity and verify the inference output on the fly. We propose a novel sensitivity analysis that identifies the most vulnerable DNN layers to the fault-injection attack. The DNN signature is constructed by encoding the weights in vulnerable layers using a low-collision hash function. During DNN inference, new hashes are extracted from the target layers and compared against the ground-truth signatures. AccHashtag incorporates a lightweight methodology that allows for real-time fault detection on embedded platforms. We devise a specialized compute core for AccHashtag on field-programmable gate arrays (FPGAs) to facilitate online hash generation in parallel to DNN execution. Extensive evaluations with the state-of-the-art bit-flip attack on various DNNs demonstrate the competitive advantage of AccHashtag in terms of both attack detection and execution overhead.

CCS Concepts: • **Computer systems organization → Embedded systems**; **Reliability**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: Deep Learning, Fault-injection, Bit-flip attack, Hashing, Embedded Systems

1 INTRODUCTION

Deep Neural Networks (DNNs) have enabled a transformative shift in various applications ranging from natural language processing and computer vision to healthcare and autonomous driving. With the deep integration of autonomous systems in safety-critical tasks, model assurance and decision robustness have gained imminent importance [7, 8]. Although DNNs demonstrate superb accuracy in controlled settings, it has been shown that they are particularly vulnerable to fault-injection attacks. Recent work [5, 20] demonstrates how changing a few bits of the victim DNN's weights can reduce the classification accuracy to below random guess. These malicious bit flips have been realized in DNN accelerators via rowhammer attacks on the DRAM containing the model weights [31].

In response to bit-flip attacks, prior work suggests adding specific constraints on DNN weights during training such as binarization [23], clustering [4], or block reconstruction [14]. Adding such constraints increases the number of bit-flips required to deplete the inference accuracy, however, they do not entirely mitigate the threat. Additionally, the proposed constraints often severely affect the underlying DNN's test accuracy. Other work [15, 16] propose to use machine learning (ML) based techniques where a simpler model is trained to detect faults in the victim DNN. However, their detection rate and false positive rate are bound by the accuracy of the ML-based detector. To ensure DNN robustness, it is crucial to augment autonomous systems with an online fault

Authors' address: Mojan Javaheripi, mojan@ucsd.edu; Jung-Woo Chang, juc023@ucsd.edu; Farinaz Koushanfar, farinaz@ucsd.edu, University of California San Diego, USA.

1550-4832/2022/8-ART
https://doi.org/10.1145/3555808

detection strategy that delivers strict performance guarantees. To the best of our knowledge, none of the earlier works provide the needed detection strategy.

We propose AccHashtag, a highly accurate real-time fault detection methodology for DNNs deployed in embedded applications. AccHashtag is the first method to provide strict statistical bounds on fault detection performance and deliver 0% false positive rate. AccHashtag extracts a unique signature from the benign DNN prior to deployment. At runtime, the signature is used to validate the integrity of the DNN and verify the inference output on the fly. We propose to leverage a low-collision hashing scheme, called the Pearson hash, to extract an 8-bit signature from the pertinent weights in each DNN layer. Our hash-based signature extraction delivers several benefits: (1) hash-based integrity check enables accurate fault detection that is robust to false alarms. (2) The hash algorithm is devised particularly for low-overhead execution on commodity processors. We design a customized FPGA core for hash generation and verification that works alongside the co-processor that hosts the target DNN. Concurrent with DNN execution, the weights are streamed to the FPGA core which then generates the hash signature. We further optimize the streaming size to maximally overlap the latency of hash generation core with the latency of communication through the underlying advanced extensible interface (AXI) with the host central processing unit (CPU).

There exist an inherent trade-off between fault detection performance and the storage/runtime overhead that is determined by the number of DNN layers used for signature extraction. To balance this trade-off, we propose a novel sensitivity analysis scheme that identifies the most vulnerable layers within the DNN to be used for signature extraction. This, in turn, leads to an extremely lightweight detection methodology that incurs negligible storage and runtime, making it amenable for use in resource-constrained embedded environments. Notably, our sensitivity analysis enables AccHashtag to achieve a 100% detection rate using as few as one layer for hash extraction.

Our detection strategy is compatible with the challenging threat model where the attacker has full control over the DRAM to freely select the location and number of bit flips. In addition, the attacker has full knowledge of the underlying detection algorithm, i.e., the hash function. To calibrate AccHashtag detection, the user does not require access to any labeled data, fine-tuning, or model training. The user only chooses a secret reordering rule to generate the input for the hash function from the DNN layer weights. Using the reordering rule, the hash signatures can be robustly extracted from the DNN at runtime without the attacker's interference.

We validate the effectiveness of AccHashtag by performing extensive experiments on various DNN architectures and visual datasets. The evaluated DNNs are injected with the state-of-the-art progressive bit-flip attack [20]. We show that AccHashtag achieves a 100% detection rate with 0 false alarms while incurring $< 1.3KB$ storage and $< 1\%$ runtime compared to DNN inference on an embedded graphics processing unit (GPU). When using our customized hash computation core on FPGA, the runtime can be further decreased by an average of 2.1×, thereby enabling online signature generation and verification alongside DNN inference. Our proposed methodology outperforms prior art across all benchmarks both in terms of attack detection and algorithm execution overhead. Compared to best prior work, AccHashtag shows orders of magnitude faster execution and lower storage. In summary, the contributions of AccHashtag are as follows:

- Introducing AccHashtag, the first framework for online detection of DNN fault-injection attacks with provable guarantees on performance.
- Constructing a novel signature generation scheme based on Pearson hash which enables low-overhead and highly accurate fault detection.
- Providing lower bounds on attack detection rate using a statistical analysis of hash collision.
- Devising a sensitivity analysis to identify vulnerable layers within any given DNN architecture. AccHashtag automatically finds DNN layers with a high probability for attack and tailors the fault detection to those layers.

- Designing an FPGA core for hash generation which enables high-throughput DNN integrity validation.

An earlier version of AccHashtag was presented in [6]. In this article, we extend our framework by: (i) Devising a custom hardware accelerator that enables real-time fault-injection detection using hashing (Section 4.4), (ii) extending our evaluations on new benchmark models with higher complexity (Sections 5.1, 5.2, 5.3). Notably, we provide the first analysis of bit-flip attacks on Transformers, and (iii) providing more analysis and discussions about the various design choices of AccHashtag, specifically the proposed layer sensitivity measurement (Section 5.2).

2 BACKGROUND AND PRIOR WORK

2.1 Bit-Flip Attack

Recent work has developed various fault-injection techniques [10, 27, 30] that can be utilized to alter bits stored in the DRAM memory. These techniques give rise to the plethora of attacks that take advantage of the bit-flipping tools to induce adversarial behavior in deployed DNNs. Researchers have demonstrated the vulnerability of DNNs to fault-injection attacks that target model parameters. Perhaps the pioneer in this domain is [17] which alters a single parameter throughout the DNN to change the classification result. Follow-up work [5] analyzes the effect of targeted bit flips induced by the Row hammer attack on DNN accuracy. The authors perform the bit flips in the floating-point representation and show that their injected bitwise errors can lead to $> 90\%$ accuracy degradation when applied on certain DNN parameters.

Current state-of-the-art bit-flip attack [20] leverages a gradient-based progressive bit search to strategically identify the vulnerable bits in the DNN. Their attack is applied on quantized DNN parameters with the fixed-point representation. Other variants of the bit-flip attack exist which leverage a similar adaptive method to find the vulnerable bits but differ in the attack objective: rather than degrading the accuracy on all samples, authors of [21, 22] perform bit flips to misclassify certain input examples as a target class. In this paper, we direct our focus to the generic untargeted bit-flip attack [20, 31] as it provides the most general attack objective. We emphasize that AccHashtag is applicable to other attack variants as our methodology relies on signature extraction and verification. This, in turn, allows us to detect (adversarial) changes in DNN parameters regardless of the underlying attack objective.

Attack Formulation. Let us denote by $\{B_l\}_{l=1}^{L}$ the total bits from the Two's complement representation of per-layer DNN weights where l is the layer index. To maximally reduce the DNN accuracy, the attacker iteratively identifies the bit with the highest gradient $\max_{B_l} |\nabla_{B_l} \mathcal{L}|$ in each layer of the DNN. Here, $\mathcal{L}$ denotes the DNN inference loss. Once the per-layer most vulnerable bits are detected, the new loss will be measured for each candidate bit-flip. Finally, the bit that results in the maximum loss is selected and flipped. The iterative process continues until the DNN accuracy falls below the attacker's desired value.

2.2 Existing Defenses

Prior art propose various techniques to increase robustness to fault-injection attacks that occur during DNN training and execution. To thwart training-time attacks, authors of [2], propose a trust-based framework as the fault detection mechanism. The performance of this method is strongly dependant on the accuracy of the trust evaluation mechanism [28, 29]. In this paper, we direct our focus to fault injection attacks applied on the DNN's internal parameters at inference time. A high-level comparison of AccHashtag with prior works is enclosed in Table 1. In what follows, we provide more details about each method and their key differences with AccHashtag.

Several prior defenses against inference-time fault injection attacks suggest adding specific constraints to the model during training. Authors of [4] show that adding a piece-wise clustering constraint to the training objective or performing binarized training can improve resiliency. Follow-up work [14] proposes to locally reconstruct DNN weights during inference to minimize or defuse the effect of the bitwise error caused by the bit

flips. Such methods increase the number of bit flips required to reduce the victim DNN's classification accuracy. However, they do not detect or prevent fault-injection attacks. Additionally, due to the added constraints on the pertinent DNN, these methods reduce the inference accuracy of the victim model. Compared to these methods, AccHashtag does not affect the inference accuracy in any way and is able to detect the occurrence of bit flips with 100% accuracy.

Other works suggest adding an ML-based attack detection mechanism. Authors of [15] train a smaller, checker network to verify the classification results produced by the original DNN. In case of a mismatch, the task is repeated and the output of the victim DNN is accepted, which results in a low detection rate. Compared to AccHashtag lightweight detection method, the checker DNN incurs a higher computational/storage overhead and can itself be subject to fault-injection attacks. Another work [16] uses the magnitude of the gradient to find sensitive weights. The authors then train a binary classifier on the sensitive weights to find bit flips. The ML-based detection techniques are bound by the classification accuracy of the underlying detector model and thereby have lower true positive rate and higher false positive rate compared to AccHashtag. We provide a probabilistic lower bound on AccHashtag detection performance that outperforms prior work.

Most recently, authors of [13] employ checksums to detect bitwise errors in weight groups. The detection performance of the proposed methodology relies on the choice of the group size, i.e., the number of weights used to compute each checksum value. To obtain a good trade-off between detection performance and the storage/runtime overhead, the authors suggested using higher group sizes. From a probabilistic point-of-view, checksum on large groups has higher false negative rate compared to our hash-based mechanism. This is because checksum inherently overlooks specific even-numbered bit flips. As shown in our experiments, the best reported results from [13] achieve lower detection accuracy compared to AccHashtag while requiring higher storage and runtime.

Table 1. High-level comparison of AccHashtag with prior work.

	100% TPR	0% FPR	Degrade DNN Accuracy	Require DNN Training	Low Overhead	Customized Hardware
Piece-wise Clustering [4]			✓	✓		
Weight Reconstruction [14]			✓	✓		
DeepDyve [15]				✓		
Weight Encoding [16]				✓		
RADAR [13]		✓	✓			
HASHTAG [6]	✓	✓			✓	
AccHashtag	✓	✓			✓	✓

3 ACCHASHTAG METHODOLOGY

Figure 1 demonstrates the high-level overview of AccHashtag methodology for detecting fault-injection attacks in DNN parameters, i.e., bit flips. The core idea in AccHashtag is to generate a compact (ground-truth) signature from the benign DNN. This is done by generating per-layer hashes of DNN parameters prior to model deployment. The signature is then used to verify the integrity of DNN parameters during execution to validate the inference result and mitigate malicious behavior. Our detection methodology incurs minimal computation/storage overhead and is devised based on lightweight solutions to enable efficient and real-time execution in embedded systems. AccHashtag comprises two main phases to detect anomalies in DNN parameters:

Pre-processing Phase. AccHashtag preprocessing is a one-time process in which the detection mechanism is calibrated for the underlying victim DNN. There exist an inherent tradeoff between attack detection performance and the computation/storage requirement for extracting layer signatures; On the one hand, hashing all layers ensures that the detection mechanism can universally adapt to attacks in any subset of layers. On the other hand,

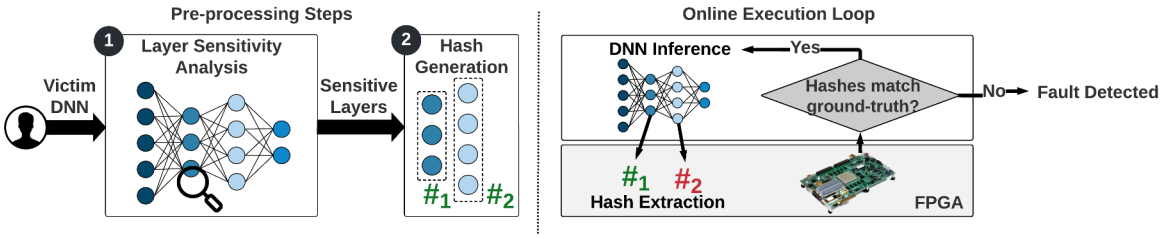

Fig. 1. Global flow of AccHashtag detection. During the pre-processing phase, we first identify sensitive DNN layers that are most prone to fault-injection attacks. We then generate a customized signature from the identified sensitive layers. During online execution, the signature is used to validate the model's integrity in real-time, parallel to conducting inference.

hash computation and storage are linear in the number of layers used for detection. We observe that various DNN layers are not equally targeted by fault-injection attacks. Motivated by this, we devise a novel sensitivity analysis scheme that models the vulnerability of DNN layers to bit-flip attacks. The top-k most vulnerable layers, called *checkpoint layers*, are then used to extract the hashes. This, in turn, allows AccHashtag to maximize detection performance under any given computation/storage budget. k is a tunable hyperparameter in AccHashtag which can range from 1 to L where L is the total number of linear layers in the victim DNN. If the user selects $k = L$, then hashes will be generated for all layers. However, as we show in our experiments (see Section 5.3.2 Figure 11), for the wide variety of evaluated models our detection rate reaches 100% when generating hashes for at most $k = 5$ layers. This is due to the ability of our sensitivity analysis to accurately locate layers with the highest probability of fault injection.

Online Execution. This recurring phase is activated when the underlying DNN is queried. During online execution, new hashes are extracted from checkpoint layers in parallel to the DNN inference. The new hashes are then validated against the ground-truth hash values from the pre-processing phase to verify the legitimacy of model parameters. Upon hash mismatch, an alarm flag is raised to notify the user that the system is compromised. The user shall then evict the deployed model and reload the ground-truth weights from the source. We accelerate the operations performed in AccHashtag's online execution phase using a customized FPGA core that interacts with the DNN's host processor.

3.1 Threat Model

In this paper, we direct our focus to fault-injection attacks that target DNN parameters, i.e., the bit-flip attack. In this scenario, the attacker has full knowledge of the victim DNN architecture and its parameters. They further know the physical address of the model parameters and have access to a subset of the data used for training the DNN. The attacker uses the data to progressively identify vulnerable weights and flip their value. This is done by performing a Row Hammer Attack (RHA) [10] on DRAM locations where the model parameter are stored [5, 31]. To keep the attack stealthy and reduce the high cost of RHA, we assume the attacker is motivated to minimize the number of flipped bits as is observed in the state-of-the-art attacks [20, 21]. As such, we do not consider random bit flips since they are shown to be ineffective in reducing DNN accuracy even with a high number of flipped weights [20, 31].

We evaluate our detection in the challenging white-box scenario where the attacker knows which layers are used for detection. He is also fully aware of the hash algorithm used for generating the per-layer signatures. However, he does not know the secret hash values and the parameter ordering used for generating the hashes. Following prior work [13], we assume the secret hashes are stored in the secure on-chip static random access memory (SRAM) which is not accessible by the attacker. Note that even when SRAM storage is not available, our

detection secrets are still immune to RHA. This is due to their low memory footprint (less than 5 KB) that makes them hard to target by RHA as shown in [5].

4 ACCHASHTAG COMPONENTS

4.1 Hash-based Signature Extraction

Hash functions generate a constant-length code value which is independent of the size of the corresponding hashed data. This property motivated us to leverage hashing as the underlying mechanism for extracting DNN layer signatures. Among the available hash functions, AccHashtag incorporates the Pearson hash [19] which operates on input streams at Byte granularity. Below we present the Pearson scheme for generating an 8-bit hash value.

Pearson Hash Formulation. The user generates a *hash table* T which contains a random permutation of integer values in the range [0, 255], i.e., $\mathbb{Z}_{256}$. For an incoming vector of length N containing Byte values $\{x_i\}_{i=1}^{N}$, the Pearson hash is defined recursively as follows:

$$h(x_1, x_2, \ldots, x_N) = T(h(x_1, x_2, \ldots, x_{N-1}) \oplus x_N) \tag{1}$$

where $\oplus$ represents the XOR operation. Since T is an arbitrary permutation of values in $\mathbb{Z}_{256}$, there exists a total of $(256)!$ hash variations for a fixed input stream. The Pearson hash can be extended to generate hashes longer than 8 bits by repeating the above process several times and concatenating the results. However, as shown in our experiments, the 8-bit Pearson hash accurately detects the state-of-the-art bit-flip attack [20].

Our hashing scheme provides several desirable characteristics that makes it particularly amenable for low-overhead detection of fault injection attacks: (1) The hash computation is well-defined for execution in 8-bit processors and embedded CPUs [19]. (2) The hashing scheme is applicable to input streams of varying lengths, thereby providing high customizability for various DNN layer configurations. (3) Pearson hash accommodates input streams with fixed-point representation which have been target to contemporary bit-flip attacks [20, 21]. Fixed-point parameter values are observed in quantized DNNs that are widely deployed in embedded systems.

Signature Generation. To extract the ground-truth signature from a benign DNN layer, we first generate a random hash table T. The pertinent layer parameters are then fed to Equation (1) as the input stream $x_1, x_2, \ldots, x_N$ to generate the secret hash of the layer. The hash input stream is generated using a user-defined secret *ordering*. An example of such ordering is shown in Figure 2. Here, the hash input stream is constructed by first traversing the layer's weight kernel in the output channel dimension. Ordering adds a zero-cost layer of complexity to AccHashtag signature generation which prevents the attacker from reproducing the per-layer secret hashes. Note that the hash input ordering does not affect AccHashtag detection performance. The user can easily choose different secret orderings for various layers or change the ordering at any time to reinforce system integrity.

4.2 Bounds on Detection Performance

In this section, we provide the worst-case performance bounds on our hash-based detection mechanism. Recall from the threat model (Section 3.1) that the attacker is not aware of the secret ordering used to generate the hashes from layer parameters. As such, even if the attacker gains full access to the Pearson hash tables, they will not be able to reproduce the ground-truth hash values. The attacker, therefore, performs the bit-flip attack without taking extra measures to preserve the ground-truth hashes. In this context, the lower bound on AccHashtag detection can be obtained by quantifying the probability of collision in our hashes. Collision occurs when multiple input streams are mapped to the same output hash. We analyze hash collision in two separate scenarios where the attacker alters 1) one or 2) more than one element of the parameter tensor in the target layer.

4.2.1 ***Single-element Alteration.*** When the attacker alters only one element in the weight block where the hash is computed, the user can detect the hash mismatch with 100% accuracy. This is due to an intrinsic collision

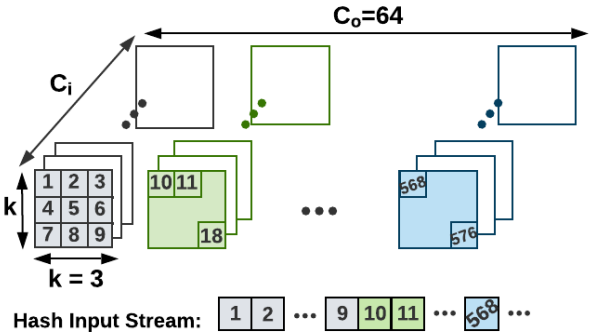

Fig. 2. Reordering parameters in an example convolution layer for generating the hash input stream. The layer parameters are the convolution weight kernels $\in \mathbb{R}^{k \times k \times C_i \times C_o}$ where k, C_i, C_o denote the kernel size, input channels, and output channels.

property for the Pearson hash: for two input streams with exactly one value difference, the probability of collision is zero when the streams are Pearson hashed.

Let us denote the altered byte value inside DNN weights by $\tilde{x}_m$. The Pearson hash operation for the first m bytes can be written as:

$$h_m = T(h_{m-1} \oplus \tilde{x}_m) \tag{2}$$

where h_i is the short notation for $h(x_1, x_2, \ldots, x_i)$. Since the first $m-1$ bytes are unaltered, the value of h_{m-1} remains constant. By changing x_m, the hash value h_m changes due to the bijective property of the hash table T. Since the remaining elements $x_i|_{i=m+1}^{N}$ are unaltered, the new hash h_m propagates through the rest of the input chain, resulting in a different final hash h_N compared to the original weight block.

4.2.2 ***Multi-element Alteration.*** In cases where the attacker changes more than one weight value in the hash block, a possibility arises that the hash mismatch caused by the earlier perturbed elements is later corrected by a subsequent perturbed weight element such that the overall hash value h_N remains unchanged. Without loss of generality let us assume only two elements are altered: $\tilde{x}_m$ and $\tilde{x}_n$ $(m < n)$. As shown previously, changing the m^{th} element, results in a new hash value that propagates through the input chain until the next changed element. Let us denote the hash value of the first $n-1$ elements in the original and altered weight blocks by h_{n-1} and $\tilde{h}_{n-1}$, respectively. To ensure the final hash value of the block remains the same, the new value of the n^{th} element $\tilde{x}_n$ needs to satisfy the following equation:

$$h_{n-1} \oplus x_n = \tilde{h}_{n-1} \oplus \tilde{x}_n \tag{3}$$

The above equation limits the number of allowed values for $\tilde{x}_n$ to only one. As such, the overall probability of obtaining the same hash after altering the bits in two elements is $\frac{1}{256} \sim 0.004$. This probability quantifies the chance of collision occurring in our hashing scheme and remains the same for any arbitrary number of elements altered bigger than one. As such, our (worst-case) lower bound on hash mismatch detection for the DNN is $\left(\frac{1}{256}\right)^{l_a}$. Here, l_a denotes the number of attacked layers where more than one weight element is flipped by the attacker.

We empirically evaluate our developed bound by performing multiple runs of hash extraction on an arbitrary input stream of length 1000. We randomly change a subset of k values within the input and measure the collision rate. As seen in Figure 3, by increasing the number of experiments, the collision probability asymptotically reaches 0.004 in all settings, which is compatible with the bound from our statistical analysis.

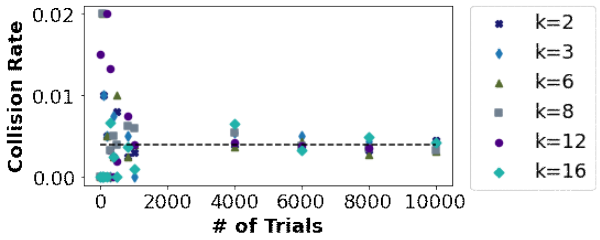

Fig. 3. Collision rate versus number of trial runs for hashing an input stream of length 1000. Each trial randomly changes a subset $k \in [2, 3, 6, 8, 12, 16]$ of message elements.

4.3 Per-layer Sensitivity Analysis

State-of-the-art fault injection attacks leverage various techniques to identify weight values that most affect the accuracy if altered. By targeting the attack towards such vulnerable weights, the attacker requires very few bit flips to degrade the accuracy of the victim DNN below random guess. Motivated by this, we devise a sensitivity analysis that accurately finds the subset of layers inside the victim DNN that are most prone to fault injection. Our sensitivity formulation is inspired by prior work in DNN pruning [18]. Specifically, we utilize Taylor expansion to model the effect of per-layer weight change on DNN accuracy as an effective measure of sensitivity.

Linear layers in DNNs comprise two key parameters, namely the weight and bias: (W, b). Let us represent the entire parameter set for a given DNN with L layers by $P = \{(W, b)_1, (W, b)_2, \dots (W, b)_L\}$ where the subscript denotes the layer index. Training the DNN is equivalent to minimizing a loss function $\mathcal{L}(D, P)$ over a corpus of data $D = (x_1, y_1), \dots, (x_d, y_d)$ where x and y correspond to input examples and their labels, respectively. To degrade a pretrained DNN's accuracy, the attacker's goal is to maximize the loss over the given dataset. Let us denote by P and $\tilde{P}$, the parameters of the DNN before and after the attack. We model the attack objective as:

$$\max_{\tilde{P}} (\mathcal{L}(D, P) - \mathcal{L}(D, \tilde{P}))^2 \tag{4}$$

We, therefore, quantify the sensitivity of each DNN parameter by the increase in loss value caused by changing it. Bit-flip attacks often alter the sign as it causes the most dramatic change in the value of the underlying parameter, thereby greatly influencing the accuracy [20]. As such, we model parameter sensitivity by altering the sign $\tilde{p} = -p$ and measuring the effect on loss. Here the lower case p represents individual weight/bias elements in the DNN. The sensitivity $S(\cdot)$ for the n^{th} parameter p_n can thus be measured as:

$$S(p_n) = (\mathcal{L}(D, P) - \mathcal{L}(D, \tilde{P}|_{\tilde{p}_n = -p_n}))^2 \tag{5}$$

Since individual computation of (5) for each weight element inside the DNN is computationally prohibitive, we leverage Taylor expansion to estimate $S(\cdot)$. For a given function $f(x)$, the first-order approximation using Taylor polynomials at point $x = a$ is given by:

$$f(x) \approx f(a) + (x - a) \times \left.\frac{\partial f}{\partial x}\right|_{x=a} \tag{6}$$

By replacing f in the Taylor expansion formula with the loss function $\mathcal{L}$, we can rewrite (5) as:

$$\mathcal{L}(D, P) - \mathcal{L}(D, \tilde{P}|_{\tilde{p}_n = -p_n}) \approx 2p_n \times \frac{\partial \mathcal{L}}{\partial p_n} \tag{7}$$

We thus measure the sensitivity of parameter p_n as:

$$S(p_n) \propto (p_n \times \frac{\partial \mathcal{L}}{\partial p_n})^2 \tag{8}$$

Note that the formula shown in (8) can be easily computed using a simple backward pass through the network to compute the first-order gradients. Once the sensitivity is obtained for each weight element, we define the sensitivity of each layer as the average over top-5 sensitivity values of its enclosing elements. We empirically explain our reason for choosing the top-5 weights by providing an analysis of the bit-flip attack in Section 5.2

4.4 Accelerating Hash Generation

To enable detection of faults in real time, we accelerate the hash computation on FPGA. This, in turn, allows for a parallel verification of weights and DNN execution. The top-level architecture of AccHashtag FPGA accelerator is shown in Figure 4. Our FPGA design communicates with the host CPU through AXI4 and AXI4-Lite interfaces. We leverage the AXI4 interface to receive the target DNN's weights in bursts. The burst reads are then fed to the input first in, first out (FIFO) buffer through the AXI master interface. The Pearson hash core interacts with the input FIFO buffer to receive the hash input stream sequentially and generate the corresponding signatures. This systolic features have low global data transfer and high clock frequency, which is suitable for large-scale parallel design, especially on FPGAs. We utilize the simpler AXI4-Lite interface to interact with the control unit and send the appropriate instructions for controlling the hash module and relevant memory transfers. AccHashtag control unit operates in three modes, namely, populating the hash table, querying the Pearson hash module for signature generation, and sending the final hash value to the host CPU. Once the hash computation concludes, the final hash value enters the output FIFO buffer and is sent back to the host CPU through the AXI master interface. We provide a break down of the utilized resources for all components in AccHashtag accelerated hash generation in Table 2. Here, the design is synthesized for a Xilinx VCU108 FPGA.

We take several measures to increase the hash generation throughput. First, we design the Pearson hash module as a specialized form of parallel computing with a deeply pipelined processes inside the hash generation loop. Using pipeline forwarding, our design fetches new data from the input FIFO buffer and calculates the hash signatures within the same pipeline stage, thereby increasing end-to-end throughput. Secondly, we take advantage of the small footprint of the hash tables to implement them entirely using 8-bit Flip-flop registers on the FPGA. This, in turn, enables very low latency accesses to the table during hash computation. Finally, we optimize the burst length for AXI reads to maximally overlap the latency of hash computation with the input stream read latency from the AXI4 Master. An optimal burst length will result in a nearly diminished cost for AXI reads, thereby increasing throughput to that of the hash Pearson hash module, with negligible increase in FPGA resource utilization.

5 EXPERIMENTS

In the following, we provide a comprehensive evaluation of AccHashtag performance along with various analyses and discussions. Section 5.1 encloses details of our benchmarked models and datasets, attack setup and implementation, as well as definitions for the utilized evaluation metrics. Section 5.2 provides an analysis of the attack profile to clarify various design choices. Finally, in Section 5.3 we report the detection performance of AccHashtag, provide comparisons with the best prior art, and analyze the storage and computation requirements of AccHashtag.

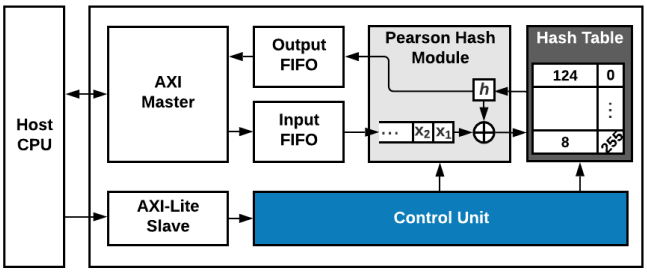

Fig. 4. Overview of AccHashtag accelerated hash generation and verification using a specialized FPGA compute kernel.

Module	BRAM	FF	LUT
Pearson Hash	-	244	962
Hash Table	2	2306	238
FIFO	-	512	580
Control Unit	-	-	1314
AXI Master	-	106	168
AXI-Lite Slave	-	144	232

Table 2. Resource utilization for AccHashtag components when synthesized on a Xilinx FPGA.

5.1 Experimental Setup

Benchmarks. We evaluate AccHashtag on two image datasets, namely, CIFAR10 [11] and ImageNet [24]. The datasets contain 10 and 1000 classes of RGB (red, green, and blue) images of dimensionality 32×32 and 224×224, respectively. We separate 20 examples from each class in the training data and create a small held-out *validation* dataset. This validation set is used to perform sensitivity analysis in the pre-processing phase.

Table 3 encloses an overview of the DNN architectures evaluated on each dataset and their baseline test accuracies with 8-bit quantization. We evaluate CIFAR10 on two DNNs, namely, ResNet20 [3] and VGG11 [26]. For ImageNet, we perform experiments on four DNNs, namely ResNet18 [3], ResNet34 [3], AlexNet [12], and MobileNetV2 [25]. We further present the first systematic study of bit-flip attacks on Transformers by benchmarking two contemporary models used in vision tasks, namely ViT [1] and DeiT [9]. We leverage the open-source code in [32] to quantize pre-trained Transformer models. As shown in Table 3, Transformers show, on average, higher robustness towards fault-injection which results in a higher number of required bit flips for degrading their accuracy.

Table 3. Overview of the evaluated benchmarks. Here, CONV, FC, and ATTN represent convolution, fully-connected, and self-attention layers, respectively. Note that each self-attention layer consists of four fully-connected layers. The baseline top-1 accuracy and the average number of bit flips are reported for 8-bit quantized DNNs.

Dataset	Model	Layers	Top-1 Acc (%)	Bit Flips
CIFAR10	VGG11	8 CONV, 3 FC	89.3	89
	ResNet20	19 CONV, 1 FC	91.9	18
ImageNet	AlexNet	5 CONV, 3 FC	55.5	21
	ResNet18	20 CONV, 1 FC	68.8	8
	ResNet34	36 CONV, 1FC	72.8	10
	MobileNet	52 CONV, 1 FC	70.3	3
	ViT	1 CONV, 12 ATTN, 1 FC	80.6	203
	DeiT	1 CONV, 12 ATTN, 1 FC	79.3	166

Attack Configuration. We leverage the open-source implementation[1] of the state-of-the-art bit-flip attack [20] to evaluate our detection. The attack batch size on convolutional neural networks is set to 128 and 64 for CIFAR10 and ImageNet benchmarks, respectively. Throughout the experiments, we repeat the attack 50 times with different initial random seeds for each of our DNN benchmarks and report the average obtained results. Each attack round

[1]Available at https://github.com/elliothe/BFA

consists of multiple iterations where one bit is flipped at each step. The iterations conclude once the DNN test accuracy falls below the random guess threshold, i.e., 10% and 0.1% for CIFAR10 and ImageNet, respectively. Due to the robustness of Transformer models to bit-flips, we consider the attack successful once the accuracy of the model falls below 0.2%. Table 3 encloses the average number of bit flips required for attacking our benchmarked DNNs in the 8-bit quantized regime.

Metrics. We use two evaluation metrics to quantify AccHashtag detection. Firstly, we define Detection Rate (DR) as the ratio of models under attack which are correctly detected by AccHashtag, as formulated in Equation (9).

$$DR = \frac{\text{\# of attacked models correctly detected}}{\text{Total \# of attack rounds}} \tag{9}$$

Secondly, we use the False Positive Rate (FPR) as the ratio of benign models mistaken for being malicious, i.e., containing a bit-flip that results in a hash mismatch.

5.2 Analysis of Design Choices

In this section, we perform an ablation study to analyze the characteristics of the bit-flip attack. We experiment with three victim DNNs with various types of (linear) layers, e.g., convolution, fully-connected, and self-attention. Specifically, we benchmark ResNet20 trained on CIFAR10 and ResNet18 and ViT trained on ImageNet. The weights in each victim DNN are quantized using a range of bitwidths. The minimum evaluated bitwidth is selected such that the classification accuracy is within 1%, 2%, and 3% of the floating-point accuracy for ResNet20, ResNet18, and ViT, respectively. For each configuration, we perform 50 runs of the bit-flip attack with different random seeds to ensure we capture the variances in the outcome. We summarize our findings below:

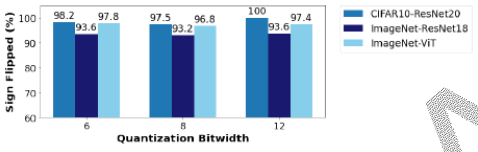

Fig. 5. Percentage of sign changes occurring during multiple runs of the bit-flip attack. The progressive bit-flip attack [20] changes the sign of the target parameter with high probability.

Sign Change. Figure 5 demonstrates the percentage of bit flips resulting in a sign change across various attack configurations. The consistent pattern among all experiments indicates that the attack significantly favors changing the sign of the target parameter. This is intuitive as flipping the sign of the underlying weight parameter can induce a dramatic change in the output of the layer. Commensurate with this finding, AccHashtag sensitivity analysis models the effect of attack as a change in the underlying parameter's sign (See Equation (5)).

Sensitivity Computation. We quantify the per-layer vulnerability to bit-flips by averaging the sensitivities of γ most vulnerable weights enclosed in each layer. Figure 6 shows the effect of various γ values on ranking DNN layers in terms of their sensitivity. On the vertical axis, the layers in each model are ordered based on their sensitivity, where a higher rank corresponds to higher sensitivity. As highlighted with the green boxes, the ordering amongst most sensitive layers remain largely the same when $\gamma \leq 10$. This is intuitive as a higher γ includes weights in the sensitivity analysis that are not prone to bit-flips, while a lower γ ensures a more targeted sensitivity analysis only for the most vulnerable parameters. For the benchmarked Transformer models, due to their inherent robustness to faults, the number of bit-flips required to reduce the accuracy is extremely large (see Table 3). For convolution-based benchmarks, however, accuracy can be downgraded with very few bit-flips. For such models, we observe that while the attack could target different or same weights within a certain layer, on average, the same layer is not targeted more than ~ 5 times. To investigate the per-layer attack concentration, we count the number of times each layer is targeted during one execution of the attack. Figure 7 shows the maximum number of bit flips occurring per layer, averaged across different attack runs for two representative

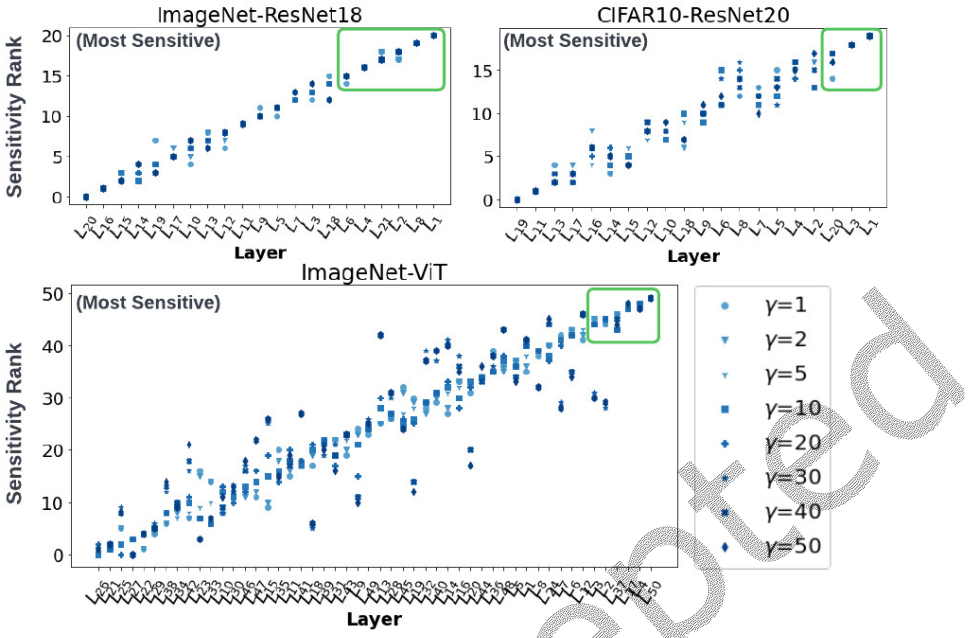

Fig. 6. Ranking DNN layers based on their vulnerability to bit-flips, defined as the average sensitivity score assigned to their γ most vulnerable weights. The most vulnerable layers (marked with green boxes), remain largely the same, when $\gamma \leq 10$.

convolutional neural networks. Using the insights from attack concentration and the analysis in Figure 6, we quantify the sensitivity of each layer as the average over its $\gamma = 5$ most sensitive weights.

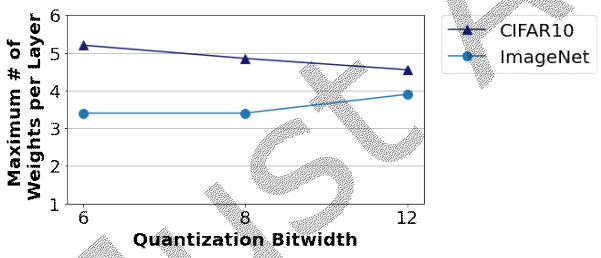

Fig. 7. Maximum per-layer attack concentration, averaged across multiple runs of the bit-flip attack to ResNet20 and ResNet18 trained on CIFAR120 and ImageNet, respectively. The progressive bit-flip attack [20] on average targets the weights in the same layer no more than ~ 5 times.

Dataset Size. We leverage a small held-out validation dataset to compute the sensitivity scores for model weights. In this section, we investigate the effect of validation dataset size, controlled by the number of held-out samples per class (n), on the sensitivity analysis. Figure 8 demonstrates the ranking of model layers in terms of sensitivity, shown for different n, where a higher rank on the vertical axis corresponds to higher sensitivity. As shown, for ImageNet benchmarks, the ranking variance is very small and the sensitivity analysis delivers consistent results even in the extreme case of $n = 1$. For CIFAR10 dataset, the sensitivity analysis is more affected by n. This is due to the small number of classes in this dataset, which results in a very small validation dataset for small n. We show the detection rate of AccHashtag versus various number of checkpoints in Figure 9. As seen for the CIFAR10 benchmark and the extreme case of $n = 1$, more checkpoints are needed to obtain 100% detection rate. However, for $n \geq 5$, hashing only the two most sensitive layers can achieve perfect detection. For the ImageNet

benchmark, $n \geq 2$ can provide 100% detection with two checkpoints. Our analysis shows an intrinsic trade-off between validation dataset size and number of checkpoint layers. When data is scarce, the sensitivity analysis may be affected, thus more checkpoint layers are needed to ensure detection.

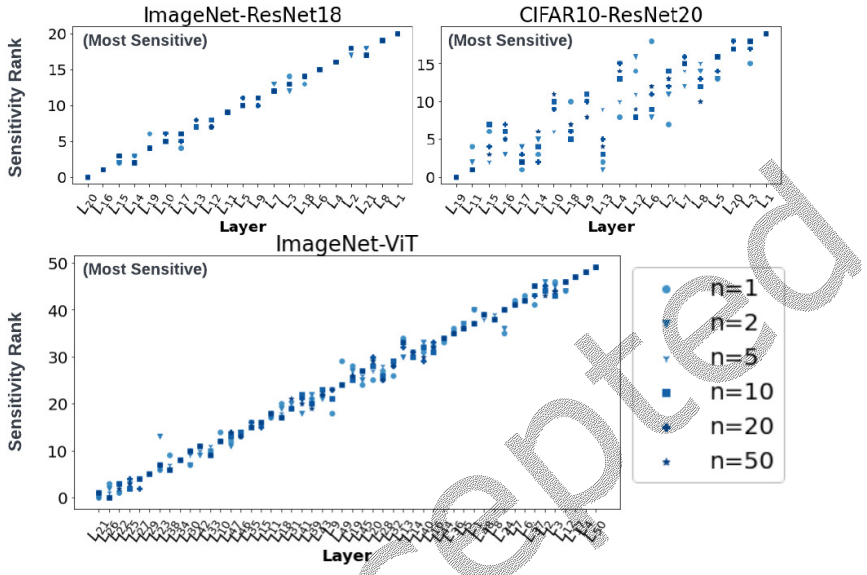

Fig. 8. Ranking DNN layers based on their sensitivity, computed using a validation dataset with n samples per class.

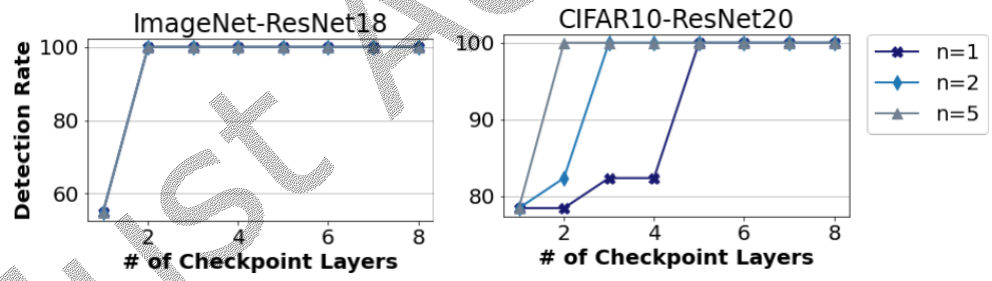

Fig. 9. AccHashtag detection rate versus number of checkpoint layers, shown for various number of per-class samples (n) used in sensitivity calculation. We omit the plot for ViT for brevity as it shows a similar trend as the ResNet18 benchmark.

5.3 AccHashtag Performance

5.3.1 ***Sensitivity Analysis.*** In this section, we showcase the stand-alone performance of AccHashtag sensitivity analysis. We benchmark the ResNet20 model on CIFAR10 to evaluate the effectiveness of our proposed method in finding the vulnerable layers within a DNN. Figure 10 demonstrates the sensitivity score assigned to each layer of the model versus the number of per-layer bit flips occurring across 50 runs of the attack. All values are normalized by the total summation. As seen, there exists a correlation between the sensitivity score

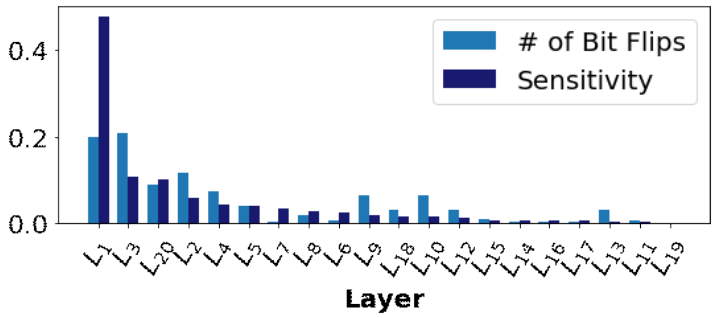

Fig. 10. Per-layer sensitivity scores assigned by AccHashtag versus the number of per-layer bit-flips. All values are normalized and sum to 1. Results are gathered across 50 runs of the bit-flip attack on the ResNet20 DNN trained with CIFAR10 dataset.

and the number of times the pertinent layer has been subject to attack; most attacks occur in layers 1, 7 which are also the most sensitive layers found by AccHashtag. Below, we provide a thorough evaluation of end-to-end AccHashtag execution.

5.3.2 ***Detection Performance***. We leverage our sensitivity analysis to rank DNN layers in the order of their attack vulnerability. The top-k most sensitive layers are then selected as checkpoints to extract hashes during the pre-processing and online phases. During online execution, if there exists *at least one* hash mismatch with the ground-truth signature among DNN layers, AccHashtag marks the model as malicious. Figures 11 and 12 demonstrates the detection performance of AccHashtag versus the number of checkpoint layers for various DNN benchmarks. For this experiment, all evaluated models are quantized with 8-bit parameters.

AccHashtag achieves a 100% attack detection rate with very few checkpoints. For the CIFAR10 benchmarks, AccHashtag detects faulty DNNs with only 1 and 2 checkpoints on the VGG11 and ResNet20 architectures, respectively. For ImageNet, AccHashtag achieves a perfect detection rate on AlexNet with only 1 checkpoint. On the more complex architectures ResNet18 and ResNet34, AccHashtag achieves 100% detection with only 2 and 3 checkpoints. For the most complex convolution neural network (CNN) benchmark, i.e., MobileNetV2 with 53 convolution and fully-connected layers, AccHashtag achieves 96.2% detection rate with 3 checkpoints and reaches perfect accuracy with 5. We further show the effectiveness of AccHashtag fault detection for large-scale Transformer-based models in Figure 12. As shown, AccHashtag successfully locates the sensitive layers that

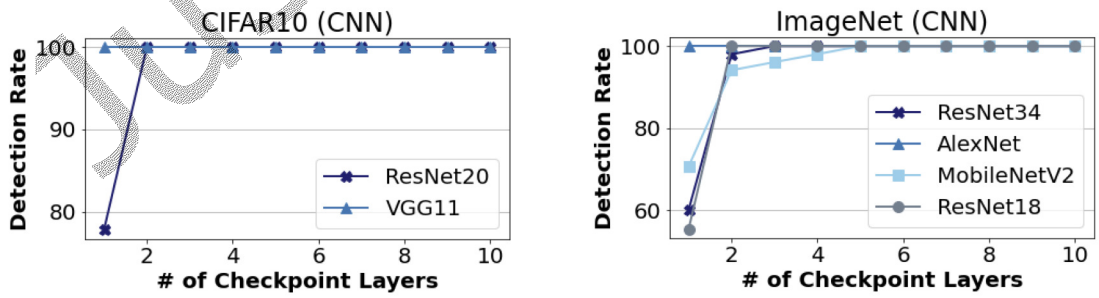

Fig. 11. AccHashtag detection rate versus the number of checkpoint layers used for signature extraction, evaluated on different victim CNNs.

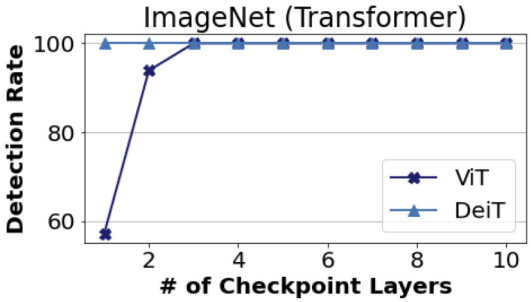

Fig. 12. detection performance of AccHashtag versus the number of checkpoint layers for various DNN benchmarks. Evaluated models are derived from the Transformer backend with self-attention layers.

are most prone to bit-flips among more than 50 layers in the Transformer benchmarks. As such, our defense can detect the occurrence of faults with 100% using only 1 and 2 checkpoint layers for the DeiT and ViT models, respectively.

The results demonstrate AccHashtag's ability to correctly find the most vulnerable DNN layers and detect fault-injections using hash signatures. Note that AccHashtag has an FPR=**0.0%**, i.e., it never mistakes benign layers for attacked ones. This is due to the fact that the hash value is constant as long as the underlying layer parameters remain intact, i.e., in the absence of bit flips.

Effect of Bitwidth. We benchmark ResNet20 and ResNet18 trained on CIFAR10 and ImageNet, respectively, and sweep the quantization bitwidth of the victim DNN. Figure 13 demonstrates the effect of DNN bitwidth on AccHashtag detection rate. While the bitwidth can affect the detection rate with only one checkpoint, it can be observed that AccHashtag becomes agnostic to the underlying bitwidth with more than 2 checkpoints. For > 2 checkpoints, AccHashtag consistently achieves a detection rate of 100%. The same trend can be observed for the Transformer-based ViT benchmark trained on ImageNet, where AccHashtag consistently achieves 100% detection rate when more than 2 (sensitive) layers are checked. the ability to maintain the detection rate in face of different quantization bitwidths allows AccHashtag to be globally applicable to various DNN configurations employed in embedded applications.

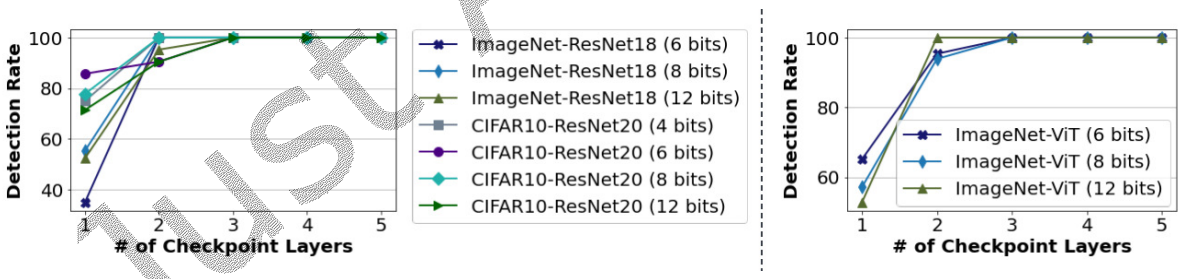

Fig. 13. Effect of victim DNN's bitwidth on AccHashtag detection rate. The legend presents the utilized datasets along with the underlying bitwidths.

5.3.3 ***Comparison with prior work.*** We compare AccHashtag with the best prior work, i.e., WED [16] and RADAR [13] in terms of detection performance and overhead. We baseline the best reported results in the original papers, i.e., the WED(2) configuration from [16], and $G = 8$ and $G = 512$ with interleaving for ResNet20 and ResNet18 from [13]. We devise two configurations for AccHashtag to enable on-par comparison with each of the prior work as follows.

Similar to AccHashtag, the proposed method in [16] checkpoints a subset of DNN layers to detect malicious models. Therefore, for best comparison with this work, we evaluate AccHashtag with the number of checkpoints set to the minimum value required to obtain 100% detection rate (see Figure 11). We call this configuration `Cfg-1`. The method in [13], however, checkpoints all layers within the DNN and reports the performance as the total number of detected bit flips. Therefore, to compare with this work, we devise `Cfg-2`, where the number of checkpoints is selected such that all bit flips are detected. For `Cfg-2`, we set the number of checkpoints to 7 and 8 for ResNet18 and ResNet20, respectively.

The comparison results are summarized in Table 4. As seen, AccHashtag provides state-of-the-art detection performance at a fraction of the storage/computation cost compared to best prior works. Compared to WED [16], AccHashtag significantly reduces the false-positive rate and achieves 100% detection rate with $FPR = 0.0\%$. Additionally, AccHashtag incurs $20 - 400\times$ lower storage footprint. Compared to RADAR [13], AccHashtag detects all bit flips within the model with 100% accuracy while incurring $3 - 4\times$ lower storage cost. We further compare AccHashtag runtime with RADAR [13]. We measure our runtime on an ARM Cortex-A57 embedded CPU. For a fair comparison, we report the normalized runtimes, i.e., relative to the inference time of the victim DNN on the target hardware. As seen, AccHashtag achieves $175 - 183\times$ faster runtime compared to [13].

We would like to emphasize that unlike [13], AccHashtag detection does not rely on the number of detected bit flips. Therefore, the setup in `Cfg-2` is purely for comparison purposes. The most representative metric for evaluating AccHashtag is the detection rate corresponding to `Cfg-1`, as explained in Section 5.1, Equation (9).

Table 4. AccHashtag comparison with best prior works WED [16] and RADAR [13]. Runtime numbers are measured on an ARM CPU and normalized by the inference time of the victim DNN.

Benchmark	Work	Detection (%)	FPR (%)	Detection Overhead	
				Storage (KB)	Runtime (%)
ResNet20	WED	96	12	47	N/A
	RADAR	97.5	0	8.2	5.27
	`Cfg-1`	**100**	**0**	**0.5**	**0.01**
	`Cfg-2`	**100**	**0**	**2.1**	**0.03**
ResNet18	RADAR	96.2	0	5.6	1.83
	`Cfg-2`	**100**	**0**	**1.8**	**0.01**
ResNet34	WED	100	4	302	N/A
	`Cfg-1`	**100**	**0**	**0.8**	**< 0.01**
MobileNet	WED	100	6	26	N/A
	`Cfg-1`	**100**	**0**	**1.3**	**< 0.01**

5.3.4 ***Storage and Computation Overhead.*** Below we provide a more detailed analysis of the storage and runtime specifications of AccHashtag detection. AccHashtag storage and computation are linear in the number of checkpoint layers: we compute and store an 8-bit secret hash per checkpoint layer. In addition, the per-layer Pearson Pearson hash tables each incur a storage cost of $256B$. The Pearson hash tables can be reused among layers, however, here we report the maximum required storage, i.e., when utilizing a unique hash table per checkpoint layer. For l checkpoint layers, the storage overhead of AccHashtag is, therefore, $O(257 \times l)B$.

To showcase the efficiency of AccHashtag detection, we measure the runtime on both an embedded Cortex-A57 CPU and an FPGA. We develop and optimize the 8-bit Pearson hash in C, which is then invoked during DNN execution to detect bit flips. We further synthesize our FPGA cores for AccHashtag on the Xilinx VCU108 development board. We utilize Vivado High-Level Synthesis to realize our FPGA design. The FPGA accelerator operates with a clock cycle of 2ns. As a baseline, we report the inference time of the victim DNN. For our CNN-based benchmarks, we report the runtimes on an embedded Jetson TX2 board which includes an ARM

Cortex-A57 CPU and an NVIDIA Pascal embedded GPU. For the large-scale Transformer-based benchmarks, we report their runtime on a server-grade Intel Xeon E5-2609 CPU and the Nvidia TITAN Xp GPU. The victim DNN is implemented and executed via PyTorch deep learning library.

Table 5 encloses the runtime and storage of AccHashtag across different benchmarks. We report AccHashtag's storage as the percentage of the memory (in Bytes) required by the victim DNN's weights. The number of checkpoints is set to the minimum value required for a 100% detection rate from Figure 11.

As evident from Table 5, AccHashtag delivers perfect detection performance while incurring a negligible storage and computation cost, making it suitable for real-time embedded DNN applications. Additionally, by leveraging our accelerated hash core on FPGA, we relieve the host CPU of hash computation. AccHashtag FPGA modules checkpoint the sensitive layers in parallel to DNN inference, enabling 1.5-2.6× faster hash generation compared to CPU execution.

Table 5. AccHashtag overhead analysis. Here, # denotes the number of utilized checkpoint layers.

Type	Benchmark	#	DNN Inference (ms)		CPU Detection		FPGA Detection
			CPU	GPU	Storage (%)	Time (ms)	Time (ms)
CNN	VGG11	1	1698.4	110.7	3e-3	0.009	0.003
	ResNet20	2	654.8	59.4	2e-2	0.012	0.005
	AlexNet	1	7957.9	240.7	4e-4	0.928	0.614
	ResNet18	2	20938.8	198.5	4e-3	0.066	0.035
	ResNet34	3	40870.6	229.7	3e-3	1.889	1.059
	MobileNet	5	2313.6	182.2	4e-2	0.020	0.007
Transformer	ViT	3	196.9	19.1	3e-3	4.692	3.127
	DeiT	1	181.3	19.5	1e-3	1.768	1.179

As discussed in Section 4.4, the hash computation is overlapped with the read latency of the hash input from AXI. To balance the latency bottleneck between these two stages, we define a design hyperparameter dubbed *TILE*, which corresponds to the length of the burst reads from AXI. Figure 14 demonstrates the relationship between design throughput (measured in number of hash computations per seconds) and *TILE* length. Using a small *TILE* will cause the memory reads to become the latency bottleneck in the design. By increasing the *TILE* value, we increase the compute capacity to match the AXI read latency, thereby increasing the overall system throughput. We empirically found $TILE = 1024$ to provide a suitable balance between AXI reads and hash computation as shown in Figure 14.

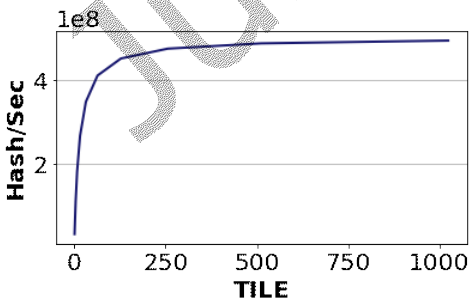

Fig. 14. System throughput as a function of the design parameter *TILE*, i.e., the burst length for AXI reads. Higher *TILE* length facilitates larger overlap between CPU-FPGA communications and hash computation, thus increasing throughput.

6 CONCLUSION

This paper presents AccHashtag, a highly accurate methodology for online detection of fault-injection attacks in DNN parameters. The core idea in AccHashtag is to extract a ground-truth signature from the benign model which is then used for verification at inference time. We extract the signatures by encoding DNN layer weights using a low-collision hash function. To minimize detection overhead, we only extract the hashes from a subset of DNN layers where the probability of attack occurrence is high. Towards this goal, AccHashtag is equipped with a novel sensitivity analysis that quantifies the vulnerability of DNN layers to bit-flip attacks. AccHashtag detection strategy provides several benefits: (1) it delivers 100% detection rate with 0 false alarms across a variety of benchmarks. (2) The proposed detection is backed up by provable performance guarantees that provide a lower bound on the detection rate. (3) AccHashtag incurs negligible storage and runtime overhead, enabling accurate fault detection on resource-constrained embedded devices. Our lightweight method and realistic threat model make AccHashtag an attractive candidate for practical deployment. Our thorough evaluations corroborate AccHashtag's competitive advantage in terms of attack detection and execution overhead.

7 ACKNOWLEDGEMENT

This work was supported in part by NSF-CNS award number 2016737, NSF TILOS AI institute award number 2112665 and the Intel PrivateAI Collaborative Research Institute.

REFERENCES

[1] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929 (2020).

[2] Anousheh Gholami, Nariman Torkzaban, and John S. Baras. 2021. On the Importance of Trust in Next-Generation Networked CPS Systems: An AI Perspective. arXiv:2104.07853 [eess.SY]

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

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

[5] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraş. 2019. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In 28th {USENIX} Security Symposium ({USENIX} Security 19). 497–514.

[6] Mojan Javaheripi and Farinaz Koushanfar. 2021. 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). IEEE, 1–9.

[7] Mojan Javaheripi, Mohammad Samragh, Gregory Fields, Tara Javidi, and Farinaz Koushanfar. 2020. Cleann: Accelerated trojan shield for embedded neural networks. In 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD). IEEE, 1–9.

[8] Mojan Javaheripi, Mohammad Samragh, Bita Darvish Rouhani, Tara Javidi, and Farinaz Koushanfar. 2020. CuRTAIL: ChaRacterizing and thwarting AdversarIal deep learning. IEEE Transactions on Dependable and Secure Computing 18, 2 (2020), 736–752.

[9] Ding Jia, Kai Han, Yunhe Wang, Yehui Tang, Jianyuan Guo, Chao Zhang, and Dacheng Tao. 2021. Efficient vision transformers via fine-grained manifold distillation. arXiv preprint arXiv:2107.01378 (2021).

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

[11] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. [n.d.]. CIFAR-10 (Canadian Institute for Advanced Research). ([n.d.]). http://www.cs.toronto.edu/~kriz/cifar.html

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems 25 (2012), 1097–1105.

[13] Jingtao Li, Adnan Siraj Rakin, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. 2021. RADAR: Run-time Adversarial Weight Attack Detection and Accuracy Recovery. arXiv preprint arXiv:2101.08254 (2021).

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

[15] Yu Li, Min Li, Bo Luo, Ye Tian, and Qiang Xu. 2020. DeepDyve: Dynamic Verification for Deep Neural Networks. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. 101–112.
[16] Qi Liu, Wujie Wen, and Yanzhi Wang. 2020. Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators. In Proceedings of the 39th International Conference on Computer-Aided Design. 1–8.
[17] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. 2017. Fault injection attack on deep neural network. In 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 131–138.
[18] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. 2019. Importance estimation for neural network pruning. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 11264–11272.
[19] Peter K Pearson. 1990. Fast hashing of variable-length text strings. Commun. ACM 33, 6 (1990), 677–680.
[20] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. 2019. Bit-flip attack: Crushing neural network with progressive bit search. In Proceedings of the IEEE/CVF International Conference on Computer Vision. 1211–1220.
[21] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. 2020. Tbt: Targeted neural network attack with bit trojan. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 13198–13207.
[22] Adnan Siraj Rakin, Zhezhi He, Jingtao Li, Fan Yao, Chaitali Chakrabarti, and Deliang Fan. 2020. T-bfa: Targeted bit-flip adversarial weight attack. arXiv preprint arXiv:2007.12336 (2020).
[23] Adnan Siraj Rakin, Li Yang, Jingtao Li, Fan Yao, Chaitali Chakrabarti, Yu Cao, Jae-sun Seo, and Deliang Fan. 2021. RA-BNN: Constructing Robust & Accurate Binary Neural Network to Simultaneously Defend Adversarial Bit-Flip Attack and Improve Accuracy. arXiv preprint arXiv:2103.13813 (2021).
[24] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. Int. J. Comput. Vision 115, 3 (Dec. 2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y
[25] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition. 4510–4520.
[26] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014).
[27] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer attacks over the network and defenses. In 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18). 213–226.
[28] Nariman Torkzaban and John S. Baras. 2020. Trust-Aware Service Function Chain Embedding: A Path-Based Approach. In 2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN). 31–36. https://doi.org/10.1109/NFV-SDN50289.2020.9289885
[29] Nariman Torkzaban, Chrysa Papagianni, and John S. Baras. 2019. Trust-Aware Service Chain Embedding. In 2019 Sixth International Conference on Software Defined Systems (SDS). 242–247. https://doi.org/10.1109/SDS.2019.8768602
[30] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic rowhammer attacks on mobile platforms. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. 1675–1689.
[31] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. 2020. Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In 29th {USENIX} Security Symposium ({USENIX} Security 20). 1463–1480.
[32] Zhihang Yuan, Chenhao Xue, Yiqi Chen, Qiang Wu, and Guangyu Sun. 2021. PTQ4ViT: Post-Training Quantization Framework for Vision Transformers. arXiv preprint arXiv:2111.12293 (2021).