# HASHTAG: Hash Signatures for Online Detection of Fault-Injection Attacks on Deep Neural Networks

Mojan Javaheripi, Farinaz Koushanfar
*Department of Electrical and Computer Engineering, UC San Diego*
mojan@ucsd.edu, farinaz@ucsd.edu

2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD) | 978-1-6654-4507-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCAD51958.2021.9643556

***Abstract*—We propose HASHTAG, the first framework that enables 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 weight bits during DNN execution by tampering with the program's DRAM memory. To detect runtime bit flips, HASHTAG extracts a unique signature from the benign DNN prior to deployment. The signature is later used to validate the integrity of the DNN and verify the inference output on the fly. We propose a novel sensitivity analysis scheme that accurately identifies the most vulnerable DNN layers to the fault-injection attack. The DNN signature is then constructed by encoding the underlying weights in the vulnerable layers using a low-collision hash function. When the DNN is deployed, new hashes are extracted from the target layers during inference and compared against the ground-truth signatures. HASHTAG incorporates a lightweight methodology that ensures a low-overhead and real-time fault detection on embedded platforms. Extensive evaluations with the state-of-the-art bit-flip attack on various DNNs demonstrate the competitive advantage of HASHTAG in terms of both attack detection and execution overhead.**

***Index Terms*—Fault-injection attacks, Deep Learning, Hashing, Embedded Systems**

## I. 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 [1], [2]. Although DNNs demonstrate superb accuracy in controlled settings, it has been shown that they are particularly vulnerable to fault-injection attacks. Recent work [3], [4] 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 [5].

In response to bit-flip attacks, prior work suggests adding specific constraints on DNN weights during training such as binarization [6], clustering [7], or block reconstruction [8]. 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 [9], [10] 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 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 HASHTAG, a highly accurate real-time fault detection methodology for DNNs deployed in embedded applications. HASHTAG is the first method to provide strict statistical bounds on fault detection performance and deliver $0\%$ false positive rate. HASHTAG 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.

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 HASHTAG 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 HASHTAG 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 HASHTAG by perform-

ing extensive experiments on various DNN architectures and visual datasets. The evaluated DNNs are injected with the state-of-the-art progressive bit-flip attack [3]. We show that HASHTAG achieves a $100\%$ detection rate with 0 false alarms while incurring $< 1.3KB$ storage and $< 1\%$ runtime compared to DNN inference on an embedded GPU. Our proposed methodology outperforms prior art across all benchmarks both in terms of attack detection and algorithm execution overhead. Compared to best prior work, HASHTAG shows orders of magnitude faster execution and lower storage.

In summary, the contributions of HASHTAG are as follows:

- Introducing HASHTAG, 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. HASHTAG automatically finds DNN layers with a high probability for attack and tailors the fault detection to those layers.

## II. Background and Prior Work

### A. Bit-Flip Attack

Recent work has developed various fault-injection techniques [11]–[13] 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 [14] which alters a single parameter throughout the DNN to change the classification result. Follow-up work [4] 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 [3] 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 [15], [16] 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 [3], [5] as it provides the most general attack objective. We emphasize that HASHTAG 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.

### B. 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 [17], 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 [18], [19]. In this paper, we direct our focus to fault injection attacks applied on the DNN's internal parameters at inference time.

Several prior defenses against inference-time fault injection attacks suggest adding specific constraints to the model during training. Authors of [7] show that adding a piece-wise clustering constraint to the training objective or performing binarized training can improve resiliency. Follow-up work [8] 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, HASHTAG 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 [9] 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 HASHTAG lightweight detection method, the checker DNN incurs a higher computational/storage overhead and can itself be subject to fault-injection attacks. Another work [10] 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 HASHTAG. We provide a probabilistic lower bound on HASHTAG detection performance that outperforms prior work.

Most recently, authors of [20] 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 [20] achieve lower detection accuracy compared to HASHTAG while requiring higher storage and runtime.

## III. HASHTAG METHODOLOGY

Figure 1 demonstrates the high-level overview of HASHTAG methodology for detecting fault-injection attacks in DNN parameters, i.e., bit flips. The core idea in HASHTAG 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. HASHTAG comprises two main phases to detect anomalies in DNN parameters:

**Pre-processing Phase.** HASHTAG preprocessing is a one-time process in which the detection mechanism is calibrated for the underlying victim DNN. There exist an inherent trade-off 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, 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 HASHTAG to maximize detection performance under any given computation/storage budget.

**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.

### A. 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

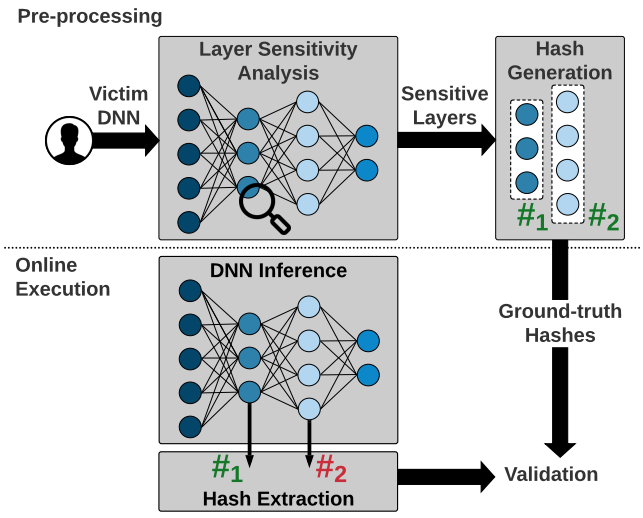


Fig. 1: Global flow of HASHTAG detection. During the pre-processing phase, we generate a customized signature from a selected subset of DNN layers. During online execution, the signature is used to validate the model's integrity in real-time.

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) [11] on DRAM locations where the model parameter are stored [4], [5]. 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 [3], [15]. 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 [3], [5].

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 [20], we assume the secret hashes are stored in the secure on-chip 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 [4].

## IV. HASHTAG COMPONENTS

### A. 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, HASHTAG incorporates the Pearson hash [21] 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:

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

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 [3].

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 [21]. (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 [3], [15]. 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 HASHTAG signature generation which prevents the attacker from reproducing the per-layer secret hashes. Note that the hash input ordering does not affect HASHTAG detection performance. The user can easily choose different secret orderings for various layers or change the ordering at any time to reinforce system integrity.

### B. 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 III-A) 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 HASHTAG 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.

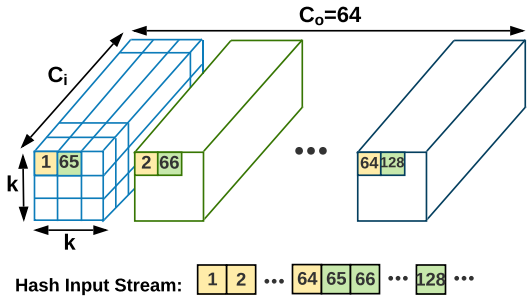


Fig. 2: Reordering parameters in an example Convolution layer for generating the hash input stream. The layer parameters are the convolution weight kernels $\in \mathbb{R}^{k \times k \times C_i \times C_o}$ where $k$, $C_i$, $C_o$ denote the kernel size, input channels, and output channels.

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.

*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 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 weight value by $\tilde{x}_m$. The Pearson hash operation for the first $m$ bits can be written as:

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

where $h_i$ is the short notation for $h(x_1, x_2, \ldots, x_i)$. Since the first $m-1$ bits 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 weight 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.

*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:

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

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.

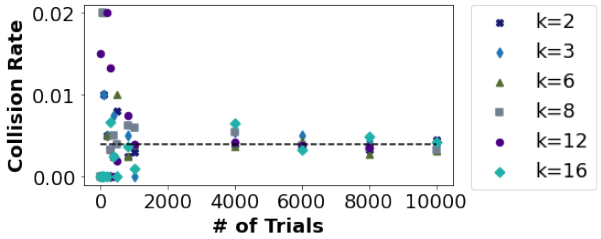


Fig. 3: Collision rate versus number of trial runs for hashing an input stream of length 1000. Each trial randomly changes a subset $k \in [2, 3, 6, 8, 12, 16]$ of message elements.

### C. 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 [22]. 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 [3]. 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 V-B

## V. Experiments

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

### A. Experimental Setup

**Benchmarks.** We evaluate HASHTAG on two image datasets, namely, CIFAR10 [23] and ImageNet [24]. The datasets contain 10 and 1000 classes of RGB images of dimensionality $32 \times 32$ and $224 \times 224$, 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 I 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 [25] and VGG11 [26]. For ImageNet, we perform experiments on four DNNs, namely ResNet18 [25], ResNet34 [25], AlexNet [27], and MobileNetV2 [28].

TABLE I: Overview of the evaluated benchmarks. Here, CONV and FC represent Convolution and Fully-connected layer, respectively. 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 | 90 |
| | ResNet20 | 19 CONV, 1 FC | 91.9 | 18 |
| ImageNet | AlexNet | 5 CONV, 3 FC | 55.5 | 25 |
| | ResNet18 | 20 CONV, 1 FC | 68.8 | 8 |
| | ResNet34 | 36 CONV, 1FC | 72.8 | 9 |
| | MobileNet | 52 CONV, 1 FC | 70.3 | 3 |

**Attack Configuration.** We leverage the open-source implementation[1] of the state-of-the-art bit-flip attack [3] to evaluate our detection. The attack batch size 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 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. Table I encloses the average number of bit flips required for attacking our benchmarked DNNs in the 8-bit quantized regime.

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

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

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.

### B. Attack Analysis

In this section, we perform an ablation study to analyze the characteristics of the bit-flip attack. We experiment with two victim DNNs, namely, ResNet20 and ResNet18, trained on the CIFAR10 and ImageNet datasets. 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% and 2% of the floating-point accuracy for CIFAR10 and ImageNet, 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:

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

**Sign Change.** Figure 4 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, HASHTAG sensitivity analysis models the effect of attack as a change in the underlying parameter's sign (See Equation (5)).

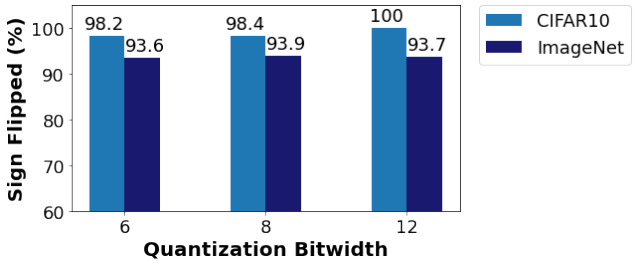


Fig. 4: Percentage of sign changes occurring during multiple runs of the bit-flip attack. The progressive bit-flip attack [3] changes the sign of the target parameter with high probability.

**Attack Concentration.** To investigate the per-layer attack concentration, we count the number of times each layer is targeted during one execution of the attack. Figure 5 shows the maximum number of bit flips occurring per layer, averaged across different attack runs. 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 $\sim 5$ times. As such, in our sensitivity analysis, we quantify the sensitivity of each layer as the average over its top-5 sensitive weights.

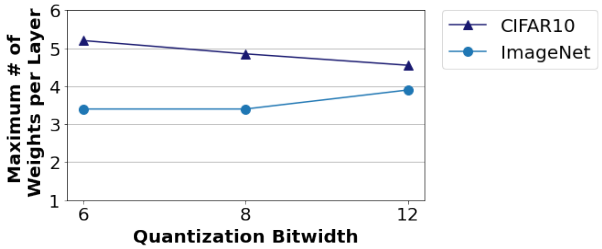


Fig. 5: Maximum per-layer attack concentration, averaged across multiple runs of the bit-flip attack. The progressive bit-flip attack [3] on average targets the weights in the same layer no more than $\sim 5$ times.

### C. HASHTAG Performance

*1) **Sensitivity Analysis**:* In this section, we showcase the stand-alone performance of HASHTAG 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 6 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,

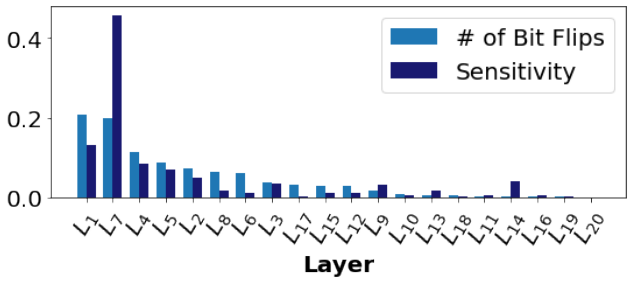


Fig. 6: Per-layer sensitivity scores assigned by HASHTAG versus the number of per-layer bit-flips. All values are normalized and sum to 1. Results are gathered across 50 runs of the bit-flip attack on the ResNet20 DNN trained with CIFAR10 dataset.

there exists a correlation between the sensitivity score 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 HASHTAG. Below, we provide a thorough evaluation of end-to-end HASHTAG execution.

*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, HASHTAG marks the model as malicious. Figure 7 demonstrates the detection performance of HASHTAG versus the number of checkpoint layers for various DNN benchmarks. For this experiment, all evaluated models are quantized with 8-bit parameters.

HASHTAG achieves a $100\%$ attack detection rate with very few checkpoints. For the CIFAR10 benchmarks, HASHTAG detects faulty DNNs with only 1 and 2 checkpoints on the VGG11 and ResNet20 architectures, respectively. For ImageNet, HASHTAG achieves a perfect detection rate on AlexNet with only 1 checkpoint. On the more complex architectures ResNet18 and ResNet34, HASHTAG achieves $100\%$ detection with only 2 and 3 checkpoints. For the most complex benchmark, i.e., MobileNetV2 with 53 convolution and fully-connected layers, HASHTAG achieves $96.2\%$ detection rate with 3 checkpoints and reaches perfect accuracy with 5.

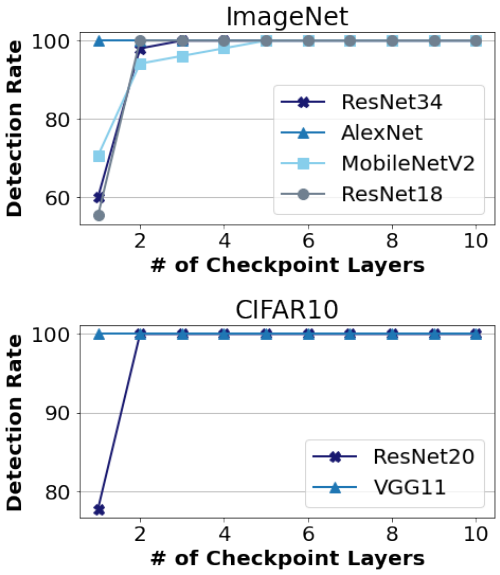


Fig. 7: HASHTAG detection rate versus the number of checkpoint layers used for signature extraction.

The results demonstrate HASHTAG's ability to correctly find the most vulnerable DNN layers and detect fault-injections using hash signatures. Note that HASHTAG has a False Positive Rate of $\mathbf{0.0}\%$, 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 and sweep the quantization bitwidth of the victim DNN. Figure 8 demonstrates the effect of DNN bitwidth on HASHTAG detection rate. While the bitwidth can affect the detection rate with only one checkpoint, it can be observed that HASHTAG becomes agnostic to the underlying bitwidth with more than 2 checkpoints. For $> 2$ checkpoints, HASHTAG consistently achieves a detection rate of $100\%$. This property allows HASHTAG to be globally applicable to various DNN configurations employed in embedded applications.

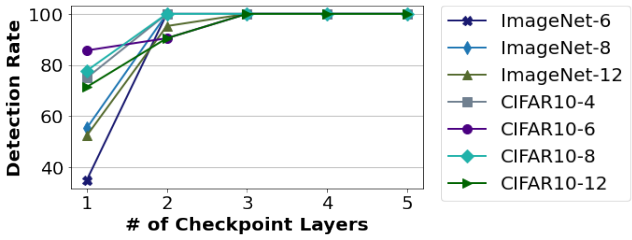


Fig. 8: Effect of victim DNN's bitwidth on HASHTAG detection rate. The legend presents the utilized datasets along with the underlying bitwidths. ImageNet and CIFAR10 evaluations are performed with ResNet18 and ResNet20, respectively.

*3) **Comparison with prior work**:* We compare HASHTAG with the best prior work, i.e., WED [10] and RADAR [20] in terms of detection performance and overhead. We baseline the best reported results in the original papers, i.e., the WED(2) configuration from [10], and $G = 8$ and $G = 512$ with interleaving for ResNet20 and ResNet18 from [20]. We devise two configurations for HASHTAG to enable on-par comparison with each of the prior work as follows.

Similar to HASHTAG, the proposed method in [10] checkpoints a subset of DNN layers to detect malicious models. Therefore, for best comparison with this work, we evaluate HASHTAG with the number of checkpoints set to the minimum value required to obtain $100\%$ detection rate (see Figure 7). We call this configuration `Cfg-1`. The method in [20], 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 II. As seen, HASHTAG provides state-of-the-art detection performance at a fraction of the storage/computation cost compared to best prior works. Compared to WED [10], HASHTAG significantly reduces the false-positive rate and achieves $100\%$ detection rate with $FPR = 0.0\%$. Additionally, HASHTAG incurs $20-400\times$ lower storage footprint. Compared to RADAR [20], HASHTAG detects all bit flips within the model with $100\%$ accuracy while incurring $3-4\times$ lower storage cost. We further compare HASHTAG runtime with RADAR [20]. 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, HASHTAG achieves $175 - 183\times$ faster runtime compared to [20].

We would like to emphasize that unlike [20], HASHTAG 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 HASHTAG is the detection rate corresponding to `Cfg-1`, as explained in Section V-A, Equation (9).

TABLE II: HASHTAG comparison with best prior works WED [10] and RADAR [20]. 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** | $\mathbf{< 0.01}$ |
| MobileNet | WED | 100 | 6 | 26 | N/A |
| | `Cfg-1` | **100** | **0** | **1.3** | $\mathbf{< 0.01}$ |

*4) **Storage and Computation Overhead**:* Below we provide a more detailed analysis of the storage and runtime specifications of HASHTAG detection. HASHTAG 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 HASHTAG is, therefore, $\mathcal{O}(257 \times l)B$.

We measure HASHTAG runtime on the Jetson TX2 embedded board with an ARM Cortex-A57 CPU and an NVIDIA Pascal GPU. We develop and optimize the 8-bit Pearson hash in C, which is then invoked during DNN execution to detect bit flips. As a baseline, we report the inference time of the victim DNN running on GPU and CPU. The victim DNN is implemented and executed via PyTorch deep learning library. Table III encloses the runtime and storage of HASHTAG across different benchmarks. The number of checkpoints is set to the minimum value required for a $100\%$ detection rate. As evident from Table III, HASHTAG delivers perfect detection performance while incurring a negligible storage and computation cost, making it suitable for real-time embedded DNN applications.

TABLE III: HASHTAG overhead analysis. Here, # denotes the number of utilized checkpoint layers.

| Benchmark | # | DNN Inference (ms) | | Detection Overhead | |
|---|---|---|---|---|---|
| | | CPU | GPU | Storage (%) | Time (ms) |
| VGG11 | 1 | 1.5e3 | 110.7 | 3e-3 | 0.1 |
| ResNet20 | 2 | 661.8 | 59.4 | 2e-2 | 0.1 |
| AlexNet | 1 | 7.9e3 | 240.7 | 4e-4 | 1.3 |
| ResNet18 | 2 | 20.9e3 | 198.5 | 4e-3 | 0.7 |
| ResNet34 | 3 | 40.8e3 | 229.7 | 3e-3 | 1.8 |
| MobileNet | 5 | 2.6e3 | 182.2 | 4e-2 | 0.1 |

## VI. CONCLUSION

This paper presents HASHTAG, a highly accurate methodology for online detection of fault-injection attacks in DNN parameters. The core idea in HASHTAG 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, HASHTAG is equipped with a novel sensitivity analysis that quantifies the vulnerability of DNN layers to bit-flip attacks. HASHTAG 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) HASHTAG incurs negligible storage and runtime overhead, enabling accurate fault detection on resource-constrained embedded devices. Our lightweight method and realistic threat model make HASHTAG an attractive candidate for practical deployment. Our thorough evaluations corroborate HASHTAG's competitive advantage in terms of attack detection and execution overhead.

## REFERENCES


[1] M. Javaheripi, M. Samragh, B. D. Rouhani, T. Javidi, and F. Koushanfar, "Curtail: Characterizing and thwarting adversarial deep learning," IEEE Transactions on Dependable and Secure Computing, vol. 18, no. 2, pp. 736–752, 2020.

[2] M. Javaheripi, M. Samragh, G. Fields, T. Javidi, and F. Koushanfar, "Cleann: Accelerated trojan shield for embedded neural networks," in 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD). IEEE, 2020, pp. 1–9.

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

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

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

[6] A. S. Rakin, L. Yang, J. Li, F. Yao, C. Chakrabarti, Y. Cao, J.-s. Seo, and D. Fan, "Ra-bnn: Constructing robust & accurate binary neural network to simultaneously defend adversarial bit-flip attack and improve accuracy," arXiv preprint arXiv:2103.13813, 2021.

[7] Z. He, A. S. Rakin, J. Li, C. Chakrabarti, and D. Fan, "Defending and harnessing the bit-flip based adversarial weight attack," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 14 095–14 103.

[8] J. Li, A. S. Rakin, Y. Xiong, L. Chang, Z. He, D. Fan, and C. Chakrabarti, "Defending bit-flip attack through dnn weight reconstruction," in 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 2020, pp. 1–6.

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

[10] Q. Liu, W. Wen, and Y. Wang, "Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators," in Proceedings of the 39th International Conference on Computer-Aided Design, 2020, pp. 1–8.

[11] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," ACM SIGARCH Computer Architecture News, vol. 42, no. 3, pp. 361–372, 2014.

[12] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer attacks over the network and defenses," in 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18), 2018, pp. 213–226.

[13] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016, pp. 1675–1689.

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

[15] A. S. Rakin, Z. He, and D. Fan, "Tbt: Targeted neural network attack with bit trojan," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 13 198–13 207.

[16] A. S. Rakin, Z. He, J. Li, F. Yao, C. Chakrabarti, and D. Fan, "T-bfa: Targeted bit-flip adversarial weight attack," arXiv preprint arXiv:2007.12336, 2020.

[17] A. Gholami, N. Torkzaban, and J. S. Baras, "On the importance of trust in next-generation networked cps systems: An ai perspective," 2021.

[18] N. Torkzaban, C. Papagianni, and J. S. Baras, "Trust-aware service chain embedding," in 2019 Sixth International Conference on Software Defined Systems (SDS), 2019, pp. 242–247.

[19] N. Torkzaban and J. S. Baras, "Trust-aware service function chain embedding: A path-based approach," in 2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), 2020, pp. 31–36.

[20] J. Li, A. S. Rakin, Z. He, D. Fan, and C. Chakrabarti, "Radar: Runtime adversarial weight attack detection and accuracy recovery," arXiv preprint arXiv:2101.08254, 2021.

[21] P. K. Pearson, "Fast hashing of variable-length text strings," Communications of the ACM, vol. 33, no. 6, pp. 677–680, 1990.

[22] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 11 264–11 272.

[23] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[24] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," Int. J. Comput. Vision, vol. 115, no. 3, p. 211–252, Dec. 2015. [Online]. Available: https://doi.org/10.1007/s11263-015-0816-y

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.

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

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