2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-5388-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISSRE62328.2024.00036

Aspis: Lightweight Neural Network Protection Against Soft Errors

Anna Schmedding
William & Mary
akschmed@cs.wm.edu

Lishan Yang
George Mason University
lyang28@gmu.edu

Adwait Jog
University of Virginia
ajog@virginia.edu

Evgenia Smirni
William & Mary
esmirni@cs.wm.edu

***Abstract*—Convolutional neural networks (CNN) are incorporated into many image-based tasks across a variety of domains. Some of these are safety critical tasks such as object classification/detection and lane detection for self-driving cars. These applications have strict safety requirements and must guarantee the reliable operation of the neural networks in the presence of soft errors (i.e., transient faults) in DRAM. Standard safety mechanisms (e.g., triplication of data/computation) provide high resilience, but introduce intolerable overhead. We perform detailed characterization and propose an efficient methodology for pinpointing critical weights by using an efficient proxy, the Taylor criterion. Using this characterization, we design Aspis, an efficient software protection scheme that does selective weight hardening and offers a performance/reliability tradeoff. Aspis provides higher resilience comparing to state-of-the-art methods and is integrated into PyTorch as a fully-automated library.**

***Index Terms*—Soft Errors, ML Classification Models, Software Protection, Error Resilience**

I. Introduction

Convolutional neural networks (CNNs) are ubiquitous, especially in image-based tasks such as image classification and object detection. When used in safety critical applications, such as self-driving cars or adaptive driver assistance systems, their reliability requirements are strict. Autonomous vehicles typically process a continuous inflow of telemetry from cameras, radar, and other sensors to accurately perceive the environment surrounding the vehicle [1]. Consequently, classification accuracy is critical as it affects system safety requirements that are in place to avoid hazards and accidents.

Significant portions of CNNs typically reside in DRAM that is susceptible to soft errors (i.e., transient hardware faults) due to cosmic radiation [2], shrinking transistors, and operating under low voltage [3]. With SEC-DEC error-correction coding (ECC), single-bit faults are correctable and double-bit faults are detectable. A double-bit fault is detected as DUE (Detected Uncorrectable Error) and the program stops. This is not advisable for safety critical applications. While past studies have shown that double- and multi-bit faults are rare comparing to the ubiquitous single-bit faults [4], multi-bit faults in recent DRAM technologies (DDR4) are shown to be as ubiquitous as single-bit ones [5] and can disrupt system operation [6]. Error correction codes such as Chipkill [7] can correct single- and multi-bit faults but at prohibitive cost.

In the specific domain of image recognition using CNNs, a triggered multi-bit fault in DRAM during inference can be masked if the resulting classification is not altered, or a well-trained network may give an unexpected erroneous misclassification. In certain domains such as automotive or healthcare applications, misclassifications can be catastrophic.

	OpenPilot [8]	**Apollo** [9]
# Convolution Layers	70	413
# Weights	5,811,616	74,349,986
# Single-Bit Fault Sites	185,971,712	2,379,199,552
# Double-Bit Fault Sites	5,765,123,072	73,755,186,112
# Triple-Bit Fault Sites	172,953,692,160	2,212,655,583,360

TABLE I: Fault space of the convolution layers in OpenPilot and Apollo. Multiple neural networks are used in these systems (e.g., 19 NNs in Apollo). These are the collective numbers.

In this paper we shed light onto *neural network resilience during inference in the presence of bit flips (soft errors) on the weights of the convolution layers* of neural networks. CNNs are ubiquitous in autonomous vehicle (AV) software and are used for perception and control. Table I illustrates the fault space of Openpilot, an adaptive driver assistance system (ADAS) [8] and Baidu Apollo [9]. The table shows that exhaustive exploration of the soft error space in AV software is not feasible: the number of fault sites is beyond the order of billions, requiring thousands of years of experiments. Clearly, there is a need for an efficient approach to *identify the location* of critical faults and understand their impact.

Here, we aim to solve a piece of this puzzle: since CNNs are important parts of larger software, we aim to identify their resilience to soft errors in the weights of filters in their convolution layers. A broader analysis of the effect of soft errors on other components such as input buffers and activation layers is outside the scope of this work.

The first research question is to identify the relative importance of weights: are bit flips that occur in certain weights more important for classification than others? Identification of such weights even in stand-alone CNNs of smaller scale comparing to those of Table I is still very challenging. Consider the VGG19 network [10] that has 16 convolution layers with a total of 20+ million weights that are typically saved as 32-bit floating point values. Exhaustive analysis of such a vast fault to evaluate the effect of a single-bit flip requires 640 million experiments, this number grows to more than 19 billion for multi-bit flips, see Table II.

Since pinpointing vulnerable weights via exhaustive search is infeasible, we evaluate the relation of "weight impor-

2332-6549/24/$31.00 ©2024 IEEE
DOI 10.1109/ISSRE62328.2024.00036

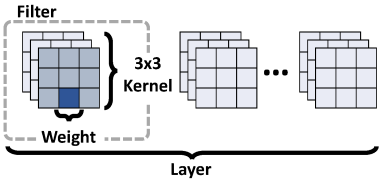


Fig. 1: Convolution layer in a CNN. Weights for a layer are a 4D tensor, depicted here as a series of 3D tensor slices.

tance" [11, 12, 13, 14] to NN reliability. Our experiments show that weights of high importance are an excellent proxy for CNN reliability assessment since a hardware fault affecting those weights almost always results in misclassification. We tested this proxy within CNNs incorporated in AV software and showed that high-importance weights can quickly identify corner cases of AV safety: soft errors on weights of high-importance can result in hazards and accidents [15].

Based on the above observation we propose Aspis[1], a neural network hardening framework that provides *protection/hardening* to high-importance weights in the presence of single- and multi-bit faults. The proportion of weights that are protected is left to the user and can be adjusted to find a desirable performance/reliability trade-off. Aspis provides superior resilience comparing to Ranger [16], a state-of-the-art framework that corrects errors in weights by restricting the ranges of the network activation layer. The overhead of Aspis is comparable to that of Ranger. In sum:

- We show that leveraging weight importance as a proxy to characterize NN reliability can be used to strike a balance between reliability and performance trade-offs. In addition, we show that weight importance can help identify corner cases that are important for safety applications.
- We perform extensive validation of weight importance as a proxy for reliability by running more than 26,000 experiments which total more than 1 billion inferences and more than 6 years of machine hours.
- We implement Aspis and integrate it into PyTorch as a fully automated library. Aspis is released publicly [17] and can be seamlessly integrated into existing NN models without additional developer effort.

II. Background

Convolutional neural networks (CNNs) are widely used in image-based tasks, including autonomous driving systems for perception and control. For image classification, input is typically an image represented as a 3D tensor (i.e., a 3D matrix) of RGB values for each pixel. The output of the CNN is a label corresponding to the image class.

Typically images are fed into the CNN model as batches. The defining feature of a CNN is the convolution layer. Convolution layers are represented as tensors of weights and are used to perform the computationally intensive convolution operation. A visualization of the weights in a convolution layer can be seen in Fig. 1. These weight tensors are typically a 4D tensor consisting of many filters. Each filter has several kernels and each kernel is a matrix of weights. In the example in Fig. 1, the convolution layer consists of several filters and each filter is made up of three 3×3 kernels. The convolution operation traverses the convolution kernels of weights over the whole input tensor to perform computation. In addition to convolution layers, CNNs have activation layers that apply a function such as ReLU, pooling layers (e.g. max pooling, average pooling), dropout layers, fully connected layers, and softmax layers, among others. The softmax layer is the final layer in classification networks which ultimately determines the most probable label for the input image.

In this paper we consider **three image classification CNNs**: VGG19 [10], ResNet50 [18], and Inception_v3 [19]. These are commonly used CNNs with pre-trained models available via PyTorch [20]. Table II shows the number of layers and weights in these CNNs. As an example, the structure of VGG19 is shown in Fig. 2. The smallest convolution layer contains 1,728 weights while the largest contains 2.3 million weights. The smallest and largest layers of ResNet50 contain 4,096 weights and 2.4 million weights. For inception_v3, the number of weights in layers ranges from 864 to 1.5 million.

Data sets. Multiple versions of the above CNNs are available through PyTorch and are pre-trained on different data sets. Here, we consider the ImageNet [21] and CIFAR-10 [22] datasets. Table III shows the test set size and the number of labels the network has to choose from for these two data sets.

A. Fault Injection and Fault Model

We use fault injection experiments to evaluate CNN reliability. We simulate commonly occurring transient faults (soft errors) in DRAM (Dynamic Random Access Memory). We assume that the standard SEC-DEC error correction is deployed in DRAM, which enables single-bit error correction. Consistent with other reliability studies [23, 24], we consider two fault models: 1) a double-bit fault model and 2) a triple-bit fault model. A fault site is located using its layer ID and weight ID. We perform fault injection in PyTorch on pre-trained PyTorch neural networks by first selecting a fault site, then flipping two (three) randomly selected bits in the float32 value for this weight, and finally updating the weight in the neural network with the new corrupted value.

To assess CNN resilience, we measure the misclassification rate of inference with the fault-injected model, i.e., among the images that are *correctly classified by the fault-free model,* we calculate the percentage of images that are misclassified due to the injected faults. We use only correctly classified images in order to distinguish between algorithmic-intrinsic errors and errors caused by bit-flips. For one fault injection run, the inference is performed on the set of *all* images that are correctly classified by the fault-free model, in other words, the fault-free model accuracy is 100% on this subset.

Note that a misclassification from the fault-injected model in *one* fault injection run only denotes the resilience of that location of the fault injection, i.e., the vulnerability of that

[1]Aspis is a wooden shield used by the infantry in Ancient Greece. Its high performance is attributed to its shape, which allows it to be supported on the shoulder and be easily used in battle.

Model	# Weights	# CONV Layers	# Single-Bit Fault Sites (ECC correctable)	# Double-Bit Fault Sites (ECC detectable; not correctable)	# Triple-Bit Fault Sites (Not detectable; not correctable)
ResNet50	23,454,912	53	750,557,184	23,267,272,704	744,552,726,528
Inception_V3	21,751,136	94	696,036,352	21,577,126,912	690,468,061,184
VGG19	20,018,880	16	640,604,160	19,858,728,960	635,479,326,720

TABLE II: Convolutional Neural Networks (CNNs) considered in this work.

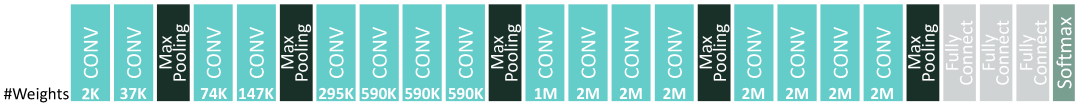


Fig. 2: VGG19 model architecture and the number of weights in each convolution layer.

Dataset	Size (Test)	Labels
Imagenet	50,000 Images	one of 1,000 categories
CIFAR-10	10,000 Images	one of 10 categories

TABLE III: Datasets.

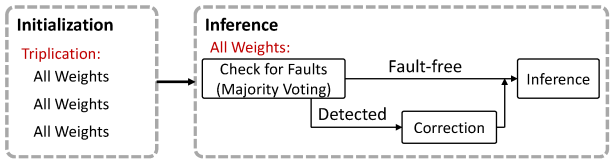


Fig. 3: Workflow of TMR: simple, but not effective.

specific site to bit flips. To evaluate the resilience of the whole CNN, one should perform exhaustive fault injection experiments on *every* fault site. We establish that this is not doable: Table II shows that the number of double- and triple-bit faults are in the order of trillions. Thus, it is impossible to perform 100% reliability coverage (i.e., evaluate all weights to assess their criticality for classification in the presence of faults). Fault injection campaigns are usually based on statistical sampling [25, 16, 26, 27] 1000 experiments (one per fault site, each randomly selected) are done to obtain results with 95% confidence intervals and ±3% error margins [26]. Statistical fault injection cannot be used to identify which are the critical weights for classification.

B. Standard Protection Mechanism

Triple Modular Redundancy (TMR) is based on data replication and is the standard hardening technique to ensure application resilience [28]. Here we briefly describe the workflow of TMR in the context of CNNs, where *all* weights are triplicated, see Fig. 3. At the initialization phase, two additional PyTorch tensors per layer are duplicated in memory to provide triplication. Next, for each batch of images in the inference phase, TMR first checks that the weight tensors are correct based on a majority voting scheme. If the tensors do not all agree, the fault is corrected via majority voting. If all three copies of the tensor differ, TMR still detects the error. Although TMR achieves high reliability, it comes with high overhead. We show the detailed overhead in the next section that motivates the idea of Aspis.

III. Motivation

The problem of protecting CNNs from soft errors includes two primary challenges: *how* to protect, and *what* to protect. In this section, we motivate Aspis by evaluating TMR. We show that TMR has overhead that is unacceptable within resource-constrained contexts, confirming that the question of *how* to protect must be addressed. We also investigate techniques for finding which parts of the CNN need protection, and show the shortcomings in the accuracy of finding these components, indicating that the question of *what* to protect must be addressed. This leads to Aspis, our software solution, discussed in Section IV.

A. TMR Implementation

Implementing basic TMR is straightforward, but has a significant downside: the runtime overhead is intolerable, see the black bars in Fig. 4 that range from 11 to 12 hours evaluating the entire dataset. On top of each bar, we report on the runtime overhead as a percentage increase without TMR. Without TMR, VGG19 takes around 10 hours due to its large layer size; Inception_v3 and ResNet50 take 4–6 hours. Figure 5 reports the memory overhead of TMR, see black graphs. Applying TMR adds 121%–299% increase in runtime (depending on the CNN and dataset) and around 240MB memory overhead, rendering TMR impossible in environments with strict latency requirements, even though its memory overhead is acceptable.

Motivation #1: The standard TMR implementation introduces tremendous runtime overhead.

B. TMR-Sparse Tensor Implementation

In order to remedy the high runtime overhead problem of TMR, we alter the implementation to utilize the *sparse tensors* provided by PyTorch to a runtime-optimized protection that we call *TMR-sparse*. The original tensor of weights remains a dense tensor, but the two additional copies are sparse tensors populated with non-zero weights utilizing the coordinate format for sparse tensors in PyTorch. Applying TMR-sparse shows significant improvement: the runtime overhead is dramatically improved, see the red bars in Fig. 4. While the latency of TMR-sparse is acceptable, it regretfully suffers from high memory overhead (around 2.2 GB, which is about 9x the memory overhead of TMR), see red bars in Fig. 5. TMR-sparse suffers from memory overhead here because it can not efficiently take advantage of the sparsity when performing full triplication. There is a natural tradeoff between the sparse and dense tensor formats – additional information stored for sparse tensors is small when the number of elements in the sparse tensor is small, but when the number of elements in

Model	Min	25%	50%	75%	Max	Min 100 Cutoff	Max 100 Cutoff
VGG19	$1.73 * 10^{-9}$	$4.89 * 10^{-3}$	$1.04 * 10^{-2}$	$1.80 * 10^{-2}$	1.14	$1.04 * 10^{-7}$	0.52
Inception_v3	0	$5.21 * 10^{-3}$	$1.14 * 10^{-2}$	$2.11 * 10^{-2}$	1.27	0	0.66
ResNet50	0	$4.01 * 10^{-3}$	$8.72 * 10^{-3}$	$1.56 * 10^{-2}$	$9.88 * 10^{-1}$	0	0.48

TABLE IV: The distribution of weight magnitude per model for the ImageNet dataset.

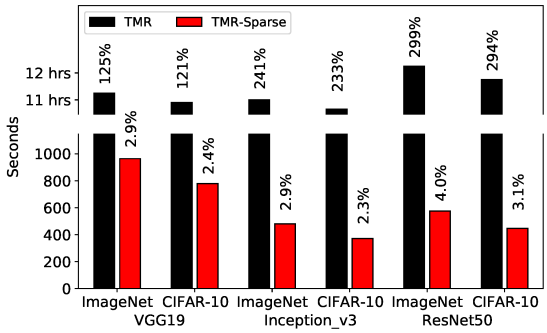


Fig. 4: Runtimes for TMR and TMR-sparse. Runtime % overhead versus the CNN execution without protection is also given. Note that we evaluate the entire dataset as input, i.e., we do not report on the inference of a single image or batch.

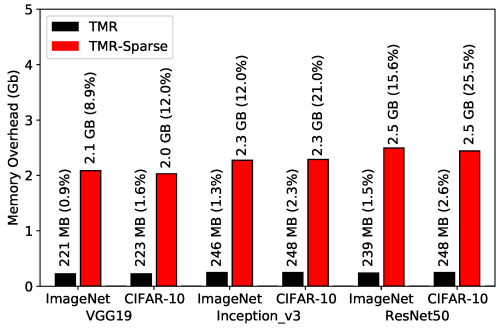


Fig. 5: Memory overhead (additional memory needed) of TMR and TMR-sparse versus the CNN without protection.

the sparse tensor is very large, this extra information degrades memory performance. Therefore, to reduce memory overhead, we cannot blindly triplicate to take advantage of the useful properties of sparse tensors.

Motivation #2: TMR and TMR-sparse result in unacceptable performance overheads.

C. Weight Magnitude and Weight Importance

Having established that protecting the entire set of CNN weights results in a low-performance solution, we entertain whether it is reasonable to protect only the portion of the most important weights for classification. We therefore need a fast but efficient proxy to identify critical weights.

It is well-known that the weight value, especially its magnitude, contributes to CNN accuracy and resilience [29]. Weight magnitude is an easy-to-characterize choice for evaluating the importance of weights. We investigate if this is a good metric for determining the criticality of weights. We perform a brief characterization of the weight magnitude, see Table IV for the distribution of weight magnitudes in CNNs.

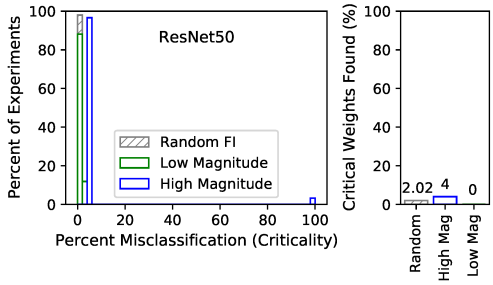


Fig. 6: Fault injection based on weight magnitude for Resnet50. Magnitude is insufficient for finding critical weights.

As a motivating example, we select the top-100 high-magnitude weights (noted as *High Mag*) and top-100 low-magnitude weights (noted as *Low Mag*) to perform fault injection experiments. We present the fault injection results for a random sampling campaign of 1000 fault sites. Fig. 6 shows the fault injection results: most experiments result in very few misclassifications. These FI campaigns find at most 4% critical weights. Thus, weight magnitude is insufficient for identifying weights that need protection.

Motivation #3: Weight magnitude is not a good proxy for identifying critical weights.

IV. Aspis : End-to-End Hardening with Low-Overhead

We present our methodology to pinpoint the vulnerabilities in CNNs by leveraging as proxy, the *importance score* of individual weights [11, 12, 30]. Using this proxy, we classify weights as "vulnerable" when faults in them result in misclassification, versus "less vulnerable" that have low importance in classification and can consequently absorb faults. We propose to selectively protect weights ranked by their importance (i.e., those that are deemed most vulnerable) and design Aspis, an end-to-end CNN hardening framework integrated with PyTorch [17]. Aspis effectively protects CNNs from soft errors while keeping runtime and memory overhead low.

A. Weight Importance Score as a Reliability Proxy

Weight Importance Score Characterization. Bit flips in critical weights can lead to silent data corruptions, which may eventually result in misclassification. Since examining every fault site in the exhaustive fault site space is not feasible, we utilize the *importance score* of each weight, a numerical value used to assess the importance of the weight to the classification output. We use this score to rank weights in order of their criticality and provide selective protection only to the most critical weights. Molchanov et al [11] calculate each filter's importance using the first-order Taylor expansion to approximate the squared difference in prediction accuracy

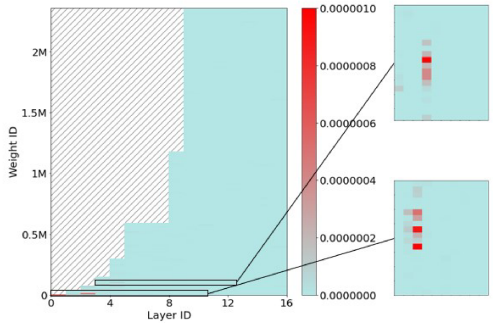


Fig. 7: Heatmap of VGG19 weight importance scores for ImageNet. The x-axis represents the layer ID and the y-axis the weight ID. The importance score of each weight ranges from low importance (light blue) to high importance (red). Most of the weights are not important; the red spots in the zoomed-in windows are weights with high importance score. The heatmap for VGG19 with the CIFAR-10 data set is similar (not shown due to lack of space).

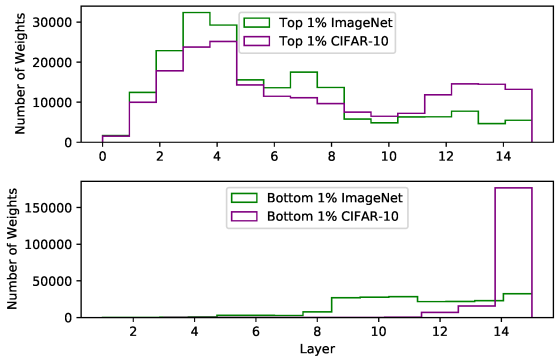


Fig. 8: Layer-wise distribution of top-1% and bottom-1% important weights in VGG19 for two different datasets.

without that filter. We modify this method to operate at a finer granularity, and calculate the importance of each *weight* instead of each filter. The importance of a weight using this expansion simplifies to the following equation:

$$I_m^{(1)}(W) = (g_m w_m)^2, \tag{1}$$

where I is the importance, W is the set of network parameters, g_m are elements of the gradient, and w_m are the weight values.

We use this modification to determine the estimation for each weight, record these values as *importance scores*, and use them to rank weights by their criticality in classification. Because this calculation is based on the weights and gradients which vary depending on the data set used for training and testing, we examine networks using both the ImageNet [21] and CIFAR-10 [22] datasets. Note that the methodology presented in this paper is independent of the method used to calculate importance scores. Our methodology can be combined with any other method for importance score calculation, such as Neuron Importance Score Propagation [12].

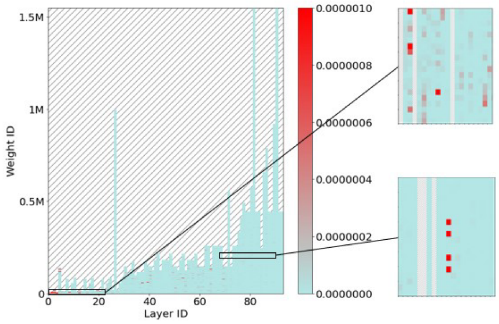


Fig. 9: Inception_v3 importance scores for ImageNet.

The distribution of importance scores across different layers for VGG19 with ImageNet is shown in Fig. 7, organized by the layer ID (horizontally) and weight ID (vertically). The light blue in the heatmap indicates that most of the weights are classified as not important. We showcase two zoomed-in windows on the right side, where the red spots are weights with high importance scores. Overall, there are very few important weights which are visible (unless zooming in).

We are also interested in the location of important weights across different layers. Fig. 8 shows the number of important weights in different convolutional layers of VGG19. From the top figure in Fig. 8 which shows the top 1% important weights, most of them are in the early layers, as also observed in the literature [31]. Note that since the first three layers are small, the number of important weights in these layers is also small. Percentage-wise, the vast majority of weights, 95.8% and 99.4% in the first layer are classified as critical for the network trained on ImageNet and CIFAR-10 respectively, further showing the importance of early layers. This layer-wise pattern is seen for both ImageNet and CIFAR-10 datasets, but there is not a complete overlap in important weight locations for networks trained on different datasets. This shows that important weight locations are dependent on both the network structure and the datasets used for training and testing.

Fig. 9 and Fig. 10 depict a heatmap and layer-wise distribution of the top-1% and bottom-1% important weights in Inception_v3 on the ImageNet dataset. Similar to VGG19, important weights are not necessarily close to one another (Fig. 9), and may be in any part of the network, but most critical weights are in the first few layers (Fig. 10). Comparing the two data sets, Inception_v3 trained on CIFAR-10 has a larger spike of critical weights in early layers than on ImageNet, showing that important weights in Inception_v3 trained on ImageNet are more spread out. However, the least important weights for Inception_v3 on both data sets are very similar and are concentrated towards the end of the network.

Fig. 11 shows the heatmap of weight importance scores for ResNet50 with the ImageNet dataset. Similar to VGG19 and Inception_v3, critical weights are spread out and difficult to locate. However, Fig. 12 shows an interesting difference between the two data sets. While the two data sets for VGG19 and Inception_v3 tend to show that the location of the most important weights are located in the same layers,

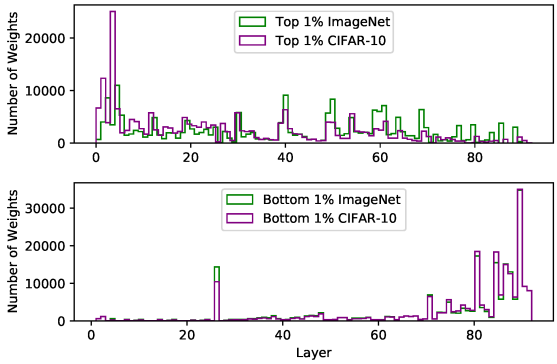


Fig. 10: Layer-wise distribution of top-1% and bottom-1% important weights in Inception_v3 for two different datasets.

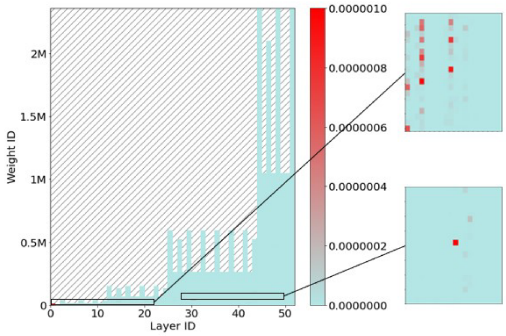


Fig. 11: ResNet50 weight importance scores for ImageNet.

the important weights in ResNet50 are distributed differently across different layers. ResNet50 trained on CIFAR-10 has a high concentration of important weights in early layers, but on ImageNet, these are concentrated later in the network. The least important weights for the network for CIFAR-10 are towards the end layers, but the least important weights for ImageNet are concentrated to three spikes. This shows that the data set used for model training has strong influence on the location of important weights.

Across all three models, there are many critical weights in the early layers, however, critical weights can be present in any layer, indicating that it is important for any protection mechanism for CNNs to account for critical weights in all layers. Models which are trained on different data sets contain different critical weights, meaning that protection mechanisms must locate and account for the critical weights present in models based both on structure and the data set for that use-case. As data sets continue to evolve, models will be re-trained with new data and it is necessary that these models with new weights can be evaluated.

Summary. Critical weights dependent on both model structure and data set. The layer location (early/later layers) is not necessarily a good indicator of the weight importance.

Importance Score as a Resilience Proxy. Here we consider three sets of fault injection experiments:

1) Random fault injection: We do 1,000 double-bit fault injection experiments on randomly selected weights from all all layers in the network, to obtain 95% confidence intervals with ±3% error margins [26];
2) Fault injection in high-magnitude weights only as in Section III.
3) Importance score-guided fault injection: here, faults are injected in the top-500 most important weights.

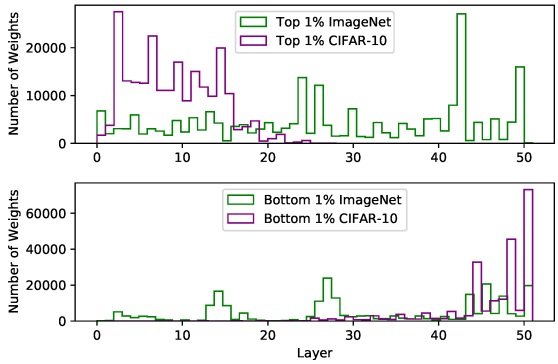
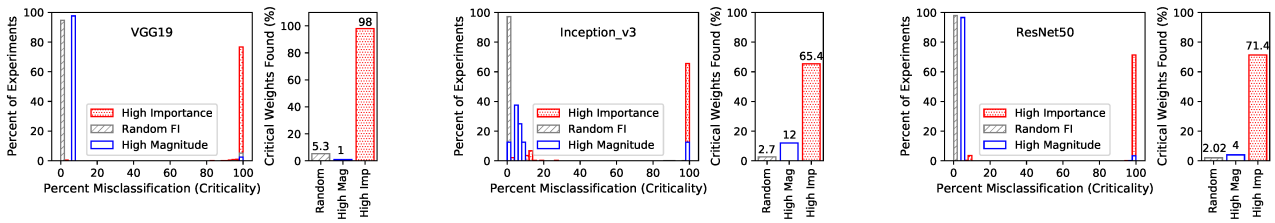


Fig. 12: Layer-wise distribution of top-1% and bottom-1% important weights in ResNet50 for two different datasets.

For all three sets of experimental campaigns, we perform inference on every image of Imagenet that is classified correctly in the golden (fault-free) run of the model. This allows us to examine misclassifications of different CNNs on the same scale. Results for fault injection in VGG19 are shown as histograms in Fig 13(a). Because the fault space is so large and so few weights are critical for resilience, random fault injection does not give a clear picture of CNN resilience. The percent misclassification (y-axis) refers to the misclassification caused specifically by fault injection. Without the injected fault, none of these misclassifications occur. As shown in the gray bar in Fig 13(a), few experiments (5.3%) find weights whose corrupted values lead to near 100% misclassification. Recalling the heatmap of weight importance scores in Fig. 7, most of the weights are not important. This explains why random sampling cannot capture the effect of critical weights. Instead, most weights selected in random fault injection lead to negligible reduction in accuracy, i.e., they are not critical. Similarly, experiments based on weight magnitude (blue bars) do not tend to find critical weights and result in less than 10% misclassifications. The red bars in Fig. 13(a) show the results of importance score-guided fault injection. Most weights (98%) from the top-500 importance scores are found to be critical, i.e., fault injections there result in misclassifications. Although weight importance score is not a perfect proxy for resilience estimation, it correctly captures more critical weights (490) than random fault injection (53) in half as many experiments. Most of the critical weights are located in the early layers of the network, however the importance score does accurately capture critical weights in later layers. In short, the critical weights in later layers indicate that a coarse-grained layer-wise protection is insufficient for protecting the network.

Similar observations are found in Inception_v3. Fault in-



(a) VGG19 fault injection results. (b) Inception_v3 fault injection results. (c) ResNet50 fault injection results.

Fig. 13: Misclassifications for random FI, magnitude-based FI, and importance-based FI for the three CNNs and the Imagenet dataset. Higher values on the x-axis mean worse accuracy.

jection results for Inception_v3 are shown in Fig 13(b). The random fault injection campaign finds few critical weights (2.7%) and the magnitude-based FI also finds an insufficient quantity of critical weights. On the other hand, the importance score-guided fault injection corroborates the claim that the weight importance score can be used as a efficient proxy for critical weights. Of the weights classified as critical using importance scores, 65.4% are indeed found to be critical. Interestingly, 23% do not have near 100% misclassification and *also* do not have near 0% misclassification (These have between 5% and 28% misclassification). This means that the importance score captures moderately important weights in addition to the critical weights. These moderately important weights cannot be located with random fault injection.

Results of fault injection in ResNet50 are shown in the histogram in Fig. 13(c). The random fault injection experiments in the gray bars in Fig. 13(c) show that most experiments have near zero percent misclassification, i.e., most of these weights are not critical. Experiments based on weight magnitude (blue bars) also fail to locate many of the critical weights. A small percentage (4%) of weights are critical weights with near 100 percent misclassification. On the other hand, the importance score-guided fault injection (red) correctly classifies 71.4% of candidate weights as critical. Similar to Inception_v3, the importance score method also manages to capture some moderately important weights in addition to critical weights.

We performed the same set of experiments on the CIFAR-10 dataset. Results are not presented here due to lack of space but we confirm that they are qualitatively the same as those that are presented in Figure 13.

Summary. Random sampling is insufficient for locating critical weights, especially in the presence of a limited time and resource budget. Weight importance scores are a good proxy for detecting weights which are likely to be critical for neural network reliability.

B. Aspis : Design and Implementation

Here we present an end-to-end hardening framework, Aspis, that provides fast and efficient protection to CNNs with minimal software engineering effort in order to facilitate the development and deployment of resilient CNNs in safety-critical domains. An overview of the entire framework can be seen in Fig. 14. This framework has two primary phases: offline analysis and online protection.

Offline analysis. The offline analysis phase performs the classification of critical weights by calculating importance scores and by recording these critical weight indices for online protection. This step is performed after the model is trained but before it is used in production. This step takes the trained model and test dataset as input and generates a list of critical weight indices as output. In our implementation, we use importance scores generated by utilizing the Taylor approximation calculations, as described in Section IV-A. We stress that weight importance analysis is not reliant on the Taylor approximation; any importance score calculation may be used.

Based on the generated weight importance scores, Aspis uses a threshold to determine the level of protection. In this work, later in the evaluation section, we select 100%, 10%, and 1% as three use cases, but Aspis can protect any percentage of weights. Essentially, Aspis-x% implies that x% of the weights are protected. Such partial protection is consistent with hardening techniques in the GPU domain: partial data replication of the most frequently accessed/shared data [32] or partial replication at the thread level [33]. Note that Aspis-100% is identical to TMR-sparse which protects all weights.

Online protection. Based on the profile of critical weights generated from offline analysis, only critical weights are protected when performing inference online. This step is performed during the usage of the model in production. Similar to TMR, triplication happens at the initialization phase, but Aspis only triplicates weights with high importance scores. Moreover, for the fault checking at the beginning of each batch inference, only weights of high importance score are checked and corrected if any errors are detected. These important weights are stored in two additional sparse tensors alongside a sparse tensor mask,[2] denoting the locations of the important weights for easy comparison.

Software Implementation. The implementation of Aspis

[2]In order to compare the whole dense tensor to the sparse tensor partial copy, we apply a mask to the dense tensor to only compare the critical weights. The indices of the mask are used to filter out the critical weights in the original tensor. The mask itself is a tensor. When the mask is applied to the original tensor, it should output a sparse tensor identical to sparse copy1 and sparse copy 2. sparse_mask() is part of PyTorch.

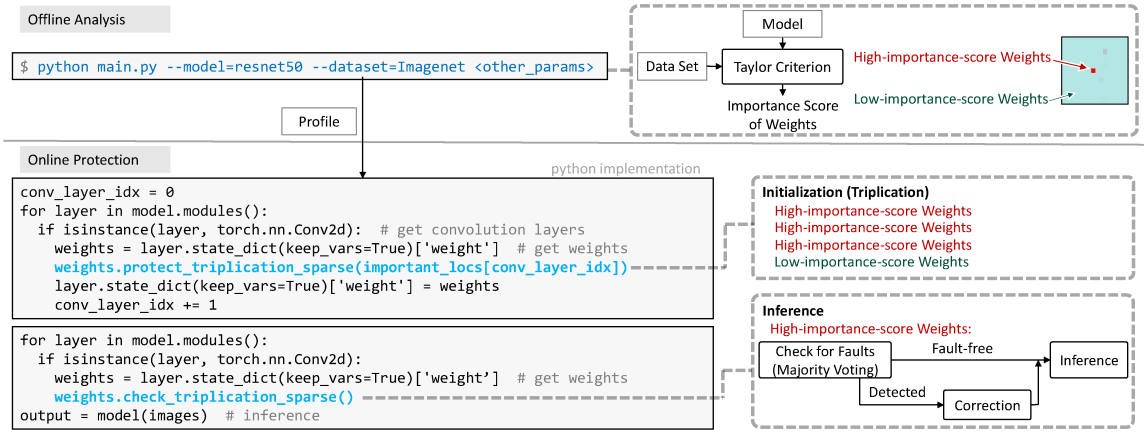


Fig. 14: Aspis workflow.

focuses on high performance and usability. The underlying implementation of the sparse tensor selective triplication is performed in the Tensor class and can be integrated into PyTorch as a new feature. In the offline analysis phase, users are able to get the profile of critical weights with one command. Then, adding the online protection typically requires several lines of code. The sample code is shown in Fig. 14. For initialization, weights are updated with the sparse tensor information by calling the *protect_triplication_sparse* function. Before inference, function *check_triplication_sparse* performs error detection and correction. No advanced knowledge of reliability and neural networks is required to apply Aspis. Aspis can be downloaded at [17].

V. Performance of Aspis

In this section, we focus on measuring the runtime overhead and memory overhead of Aspis. First, we describe our experimental setup, then present the main evaluation results.

A. Experimental Set-Up

We apply Aspis on the VGG19, Inception_v3, and ResNet50 models with the ImageNet and CIFAR-10 data sets. There are three settings, Aspis-1%, Aspis-10%, and Aspis-100% protecting the top 1%, the top 10%, and all critical weights, respectively. Aspis-100% is effectively TMR-sparse. All experiments are executed on a dedicated server with an Intel Xeon E5-2643 CPU and an NVIDIA Titan RTX GPU. We use the recommended batch size for each model (256 for VGG19 and 512 for ResNet50 and Inception_v3).

B. Resilience Comparison to the State-Of-The-Art

Ranger [16] is a state-of-the-art fault corrector that employs range restriction on neuron activation layers to protect NN models from faults. Each restriction has a pair of minimum and maximum activation values to use as bounds, which are set after profiling input data. This step is performed once, before the deployment of the protected model with Ranger. When Ranger is active, any activation values outside the ranges defined by Ranger are clipped to the profiled bound.

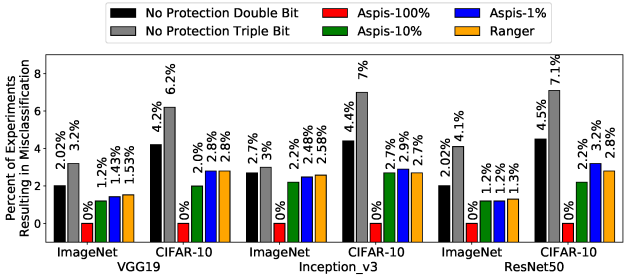


Fig. 15: Fault injection in weights selected randomly. Note that the y-axis starts from below zero, to illustrate a bar for those that achieve 0% misclassifications.

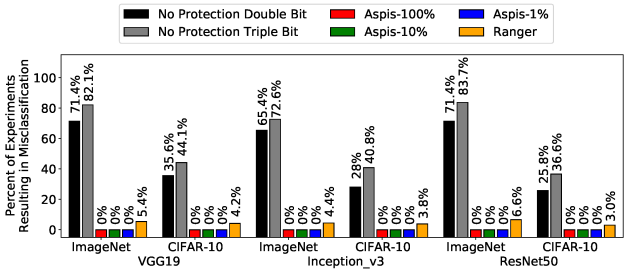


Fig. 16: Fault injection in the most important weights. Note that the y-axis starts from below zero, to illustrate a bar for those that achieve 0% misclassifications

We perform fault injection experiments to evaluate Aspis-1%, Aspis-10%, Aspis-100%, and Ranger, using three CNN models (VGG19, Inception_v3, and ResNet50) with both the ImageNet and CIFAR-10 data sets on double-bit and triple-bit fault models. We consider two fault injection scenarios: randomly selected weights and weights of high importance. The same set of faults (i.e. the same bit positions in the same weights) is used for both Aspis and Ranger. Since these experiments focus on specific weights, it allows for direct comparisons between the two protection schemes and statistical comparison is unnecessary.

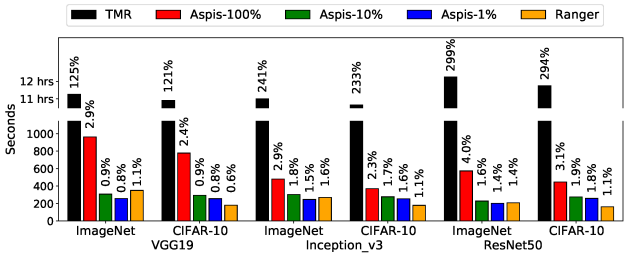


Fig. 17: Runtime overhead for protection.

Fig. 15 shows that if faults are selected randomly, then Aspis has similar success on correcting them since it only protects critical weights. Fig. 16 shows the results when injecting faults into critical weights. Since Aspis protects the weights where fault injections occur, all models have zero misclassifications. Ranger performs well overall but a small percentage (3% to 6.6%) of fault injections result in misclassifications. Both figures show that triple-bit faults result in more errors than double-bit faults, which is consistent with the literature [24]. Protection with Aspis or Ranger is exactly at the same level irrespective of the number of bit flips.

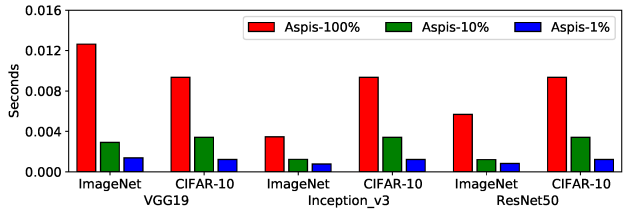
C. Runtime Overhead

The runtime overhead of Aspis is shown in Fig. 17 with a comparison across TMR and Ranger. Runtime overhead is calculated as the percent of additional time required for protection compared to the model with no protection. Note that when measuring overhead, all images in the ImageNet and CIFAR validation sets are included as input images, in contrast to the fault injection experiments which only use correctly classified images. We focus on the runtime over the whole dataset because the different steps are not performed the same number of times in practice – initialization (an expensive step) is performed only once, but we check correctness many times in the inference phase, so including the whole dataset of inferences provides a more concrete view of the total overhead cost for an application.

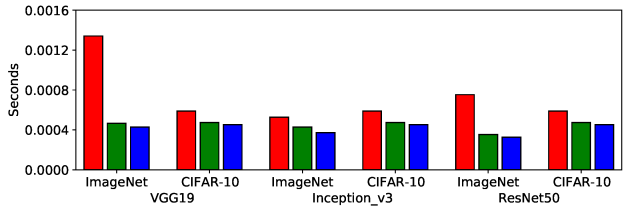
Results are presented with 95% confidence across 50 experiments each to accurately capture the runtime. Note that in Fig. 17 the error margins are less than one second, so the error bars are not visible. As we also show in Section III, TMR exhibits tremendous runtime overhead, which is more than 11 hours for all the CNN models and datasets. Recall that our benchmark is inference of the entire dataset. This is a runtime increase between 121% and 299% (see black bars in Fig. 17). Instead, Aspis and Ranger have acceptable negligible runtime overhead from 1.1% to 4% only. In general, Aspis takes advantage of the sparse tensor structure by protecting only the most critical weights. Aspis-100% is naturally more expensive since it protects everything and is therefore using a dense tensor in sparse format.

For Aspis, we also opt to a detailed breakdown of the overhead for easier comparisons and understanding. Fig. 18 presents the detailed breakdown of the runtime overhead:

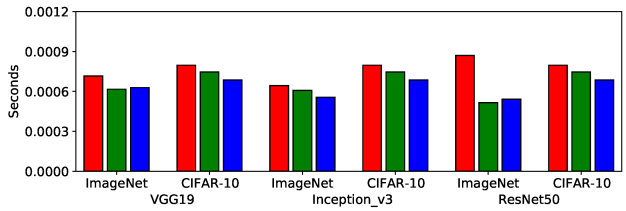
- Initialization, where weights are triplicated;



(a) Initialization overhead (per layer)



(b) Overhead for checking correctness (per layer per batch)



(c) Detection/correction overhead for protection (if triggered)

Fig. 18: Runtime overhead of Aspis: detailed breakdown.

- Checking correctness of the weights when no errors are detected;
- Checking correctness of the weights when errors are detected and corrected (if correction is possible).

Comparing these three cases, the initialization step has the highest overhead, but this is only a one-time overhead spent prior to any inference. The initialization overheads of Aspis-1% and Aspis-10% are much lower than Aspis-100% because the amount of data to be processed is different. Checking the correctness of a layer of weights is performed once per batch during model inference. The overhead of this check is shown in Fig. 18(b). Overhead is presented per layer per batch. As expected Aspis-1% and Aspis-10% have the least overhead.

If the check fails, then error detection/correction is triggered. This step is performed immediately after the check, between inferences of two batches. For a detected error and its correction, the overhead is similar regardless of the protection scheme, however protection schemes that make better use of the sparsity tend to perform better than those that cannot. For correction of an error in a sparse copy, we correct the error with the other sparse copy which is a fast operation. For correcting the main tensor, we utilize one of the copies along with the masked tensor to correct the incorrect value using subtraction. This a fast operation, although correcting the main tensor is slightly more time-consuming than correcting one of the copies. Detecting an uncorrectable error is fast, because there are no steps we can take to correct any tensors, and

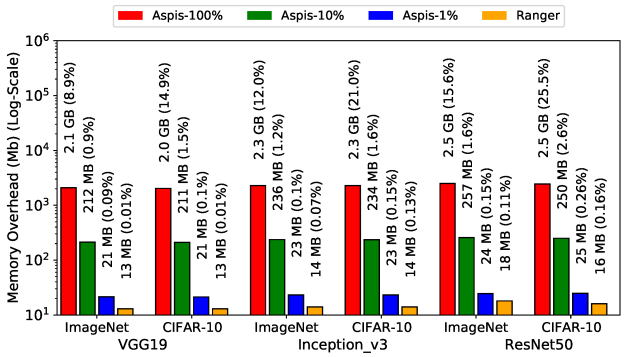


Fig. 19: Memory overhead for protection.

Protection	Runtime Overhead	Memory Overhead
Structural Coding [25]	3.5%	14.1%
Aspis-100%	4.0%	15.6%
Aspis-10%	1.6%	1.6%
Aspis-1%	1.4%	0.15%

TABLE V: Overhead comparison w/ Structural Coding [25] on the ResNet50 model with the ImageNet dataset.

performing the comparison steps alone is fast. This is clearly illustrated in Fig. 18(c), where the cost is less than 0.0008 seconds in all cases.

D. Memory Overhead

The memory overhead of protection on the three networks is depicted in Fig. 19. Memory overhead is calculated as the percent of additional memory required for protection compared to the model with no protection. The graph presents the additional memory used as a raw number but also as a percentage. Ranger has the best performance from all cases. The memory overhead for Aspis-100% is high as also discussed in Section IV-A. In resource-limited environments, memory overhead of more than 2GB is not acceptable. There are significant memory utilization improvements shown in Aspis-10% and Aspis-1%. This shows that Aspis leverages partial protection effectively and takes advantage of the sparse tensor format for efficient memory usage.

E. Comparison to Hardware-based Approaches

We perform comparisons in overhead to Structural Coding [25] which offers protection and correction of soft errors using a syndrome-based error correction strategy. Structural Coding outperforms existing state-of-the-art baselines, such as RADAR [34], MILR [35], and Chipkill [36, 37]. Differently from our method, Structural Coding offers protection of more weights in the model through their use of checksums, leaving out only a few corner cases which are not protected. Our method, Aspis, offers protection of a smaller portion of weights, except of course Aspis-100% that protects all weights of the convolution layers.

The comparison is performed on the ResNet50 model with the ImageNet data set as this is the overlapping set-up of model and dataset that is also studied in [25]. We perform our overhead evaluation per our experiment set-up in Sec. V-A. Comparisons to Structural Coding are presented in Table V. As also shown previously, Aspis-100% introduces a small runtime overhead, however its memory overhead poses a challenge since this 15.6% overhead is 2.5 GB in our experiments (see also Fig. 19). Structural Coding offers minor improvements over Aspis-100% in both runtime and memory, while also offering broader protection since it includes all weights (not just convolution layer weights). The memory overhead, while an improvement over Aspis-100%, may still be a challenge in systems that are resource-constrained. Aspis-10% and Aspis-1% offer significant overhead improvements in both runtime and memory compared to Structural Coding. Overall, Structural Coding offers excellent accuracy, but at higher overhead cost. Aspis is an efficient alternative to these existing solutions such as Structural Coding and Chipkill, specifically for reliable operation in extreme resource-constrained environments such as autonomous vehicles.

F. Case Study: Autonomous Driving

As a case study, we evaluate Aspis on an autonomous driving agent, Learning By Cheating (LBC) [38] and simulate the driving environment using an autonomous driving simulator, CARLA [39]. The model used in LBC has a ResNet34 backbone to process input images from the camera sensor and then separate convolution branches which correspond to high-level control commands (i.e. turn right, turn left, go straight at an intersection, or follow lane). The output of the model is a set of five waypoints for the vehicle to follow. For the results that we report here, we select a simple scenario: the vehicle drives along a straight road until it reaches its destination. If the vehicle crashes into an object, then the simulation terminates and an accident is recorded. Faults are injected at the beginning of the simulation. We profile important weights for LBC using the modified Taylor criterion and the Comma2k19 data set [40]. This data set contains video footage and other sensor recordings from actual vehicles driving on the road. Table VI shows the results of random faults and faults in critical weights with no protection, and shows the results from protection with Ranger and Aspis. Ranger can mitigate most faults in these simulations, however, it is necessary not to have any crashes in order for the vehicle to be safe. Aspis-1% instead offers sufficient coverage.

Faults injected	Vehicle Crash
Random Fault Injection	5.8%
Fault Injection on Important Weights	12%
Fault Injection on Important Weights + Ranger	0.4%
Fault Injection on Important Weights + Aspis-1%	0%

TABLE VI: Autonomous driving Agent LBC. The vehicle needs to have 0 crashes to be safe.

VI. Related Work

Importance scores. Importance scores rank components of the neural network by some measure of importance that is used to achieve goals such as model compression and model pruning. Various approaches are used to determine component

criticality, such as NISP [12] which determines importance by how values propagate through the neural network. Pruning methods may use the norm of the filters to assess their relative importance for the sake of model compression [13] and may also incorporate other information, such as the mean, standard deviation, activation function values [14], and kernel patterns [30]. The Taylor expansion of a component's contribution to model accuracy, in particular, may be used for assessing filter importance for neural network pruning [11]. The importance of neurons is studied through ablation in [41], however this technique cannot be applied at weight-level granularity due to the number of experiments needed (one per weight). The importance of pixels to the output class is studied in [42, 43, 44]. In particular, [43, 44] focus on determining test data set adequacy. *Differently from these works, we consider importance scores at a weight-level for reliability instead of model compression, pruning, and test data set adequacy.*

Resilience estimation and protection in neural networks. Li et al [31] use fault injection to study error propagation behaviors in DNNs and present insights for DNN design. Ibrahim et al. [45] study the impact of soft errors on Deep Residual Networks (ResNets) which are used for object recognition and classification. DeepXplore [46] and DeepTest [47] focus on the robustness and reliability testing of deep learning and autonomous driving systems as well as finding corner cases and vulnerable contexts, but not on protection solutions.

[48] performs analysis of CNNs under rowhammer attacks causing single bit-flips. Fault injections are performed using key heuristics that include sampling the validation set for faster evaluation, flipping the exponent bit only for a big perturbation impact, and sampling weights uniformly at random. Aspis uses the importance score in order to ensure that all weights are investigated for potential protection and to ensure that no classification bias is introduced from sampling from the validation set instead of using it in its entirety. Here, we consider random flips at any bit position, to capture the effect of bit flips that can happen in the wild. In summary, the heuristics used in [48] are useful for evaluating rowhammer attacks, but cannot be applied directly in our use case.

Protection efforts on NNs primarily focus on protecting the neural network in a coarse-grained fashion, e.g., kernel-level [49], layer-level [50], or feature-map-level protection [16, 49, 51]. Structural coding [25] protects DNNs through a syndrome-based error correction strategy utilizing checksums. RADAR [34] also provides error correction using checksums. MILR [35] corrects errors using the mathematical relationship between inputs, outputs, and weights. Algorithm-based fault tolerance (ABFT) can increase the reliability of object detection [52]. Adam et al. use intensive fault injection experiments to identify vulnerabilities in DenseNet201 and perform selective protection [53]. FILR [51] combines feature map duplication with full inference reruns on selected vulnerable inferences to achieve high coverage without full duplication, i.e., it estimates the vulnerability of feature maps and duplicates computation of selected feature maps by performing selective protection. Our work differs from the above because: 1) rather than adding redundant computation to detect faults, we apply selective data redundancy in candidate weights in order to correct faults with low time- and space-wise overhead, and 2) we focus on finer granularity before a feature map is computed. Because of this finer granularity, we maintain low overheads.

None of the existing work provides an automatic protection framework for low-overhead CNN protection for general platforms. *To our best knowledge, Aspis is the first end-to-end automatic framework that enables software developers to apply low-overhead protection on CNNs with minimal software engineering efforts.*

VII. Conclusions

We show that CNNs are vulnerable to faults and need to be protected given their wide applicability in safety-critical applications. Standard software protection mechanisms like TMR either consume too much memory or increase CNN runtime significantly. To address the above, we illustrated that some weights are more important than others for reliability and that protecting weights of higher importance can strike a performance-reliability tradeoff. Pinpointing these most vulnerable weights, especially in complex software toolchains, is not trivial. Aspis automates these procedures, and can be used by practitioners to make their own NN software resilient and efficient, striking a balance between reliability and performance.

Acknowledgments

This work is supported by the National Science Foundation (NSF) grants (#2402940 and #2402942) and the Commonwealth Cyber Initiative (CCI) grant (#HC-3Q24-047). Jog contributed primarily while he was with William & Mary.

References

[1] S. Jha, S. S. Banerjee, T. Tsai, S. K. S. Hari, M. B. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "Ml-based fault injection for autonomous vehicles: A case for bayesian fault injection," in *49th IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*. IEEE, 2019, pp. 112–124.

[2] V. Fratin, D. A. G. de Oliveira, C. B. Lunardi, F. Santos, G. Rodrigues, and P. Rech, "Code-dependent and architecture-dependent reliability behaviors," in *DSN*, 2018, pp. 13–26.

[3] N. Chandramoorthy, K. Swaminathan, M. Cochet, A. Paidimarri, S. Eldridge, R. V. Joshi, M. M. Ziegler, A. Buyuktosunoglu, and P. Bose, "Resilient low voltage accelerators for high energy efficiency," in *Proceedings of HPCA'19*. IEEE, 2019, pp. 147–158.

[4] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A large-scale study of soft-errors on gpus in the field," in *Proceedings of HPCA'16, Barcelona, Spain, March 12-16, 2016*, pp. 519–530.

[5] M. V. Beigi, Y. Cao, S. Gurumurthi, C. Recchia, A. C. Walton, and V. Sridharan, "A systematic study of DDR4 DRAM faults in the field," in *Proceedings of HPCA'23, Montreal, QC, Canada, February 25 - March 1, 2023*, pp. 991–1002.

[6] V. Oles, A. Schmedding, G. Ostrouchov, W. Shin, E. Smirni, and C. Engelmann, "Understanding GPU memory corruption at extreme scale: The Summit case study," in *Proceedings of the 38th ACM International Conference on Supercomputing, ICS 2024*, pp. 188–200.

[7] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," in *IBM Microelectronics division, Vol. 11. 1–23*, 1997.

[8] "OpenPilot." [Online]. Available: https://github.com/commaai/openpilot

[9] "Baidu Apollo," https://developer.apollo.auto.

[10] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

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

[12] R. Yu, A. Li, C. Chen, J. Lai, V. I. Morariu, X. Han, M. Gao, C. Lin, and L. S. Davis, "NISP: Pruning networks using neuron importance score propagation," in *Proceedings of CVPR'18*, 2018, pp. 9194–9203.

[13] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *CoRR*, vol. abs/1608.08710, 2016.

[14] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient transfer learning," *CoRR*, vol. abs/1611.06440, 2016.

[15] A. Schmedding, P. Schowitz, X. Zhou, Y. Lu, L. Yang, H. Alemzadeh, and E. Smirni, "Strategic resilience evaluation of neural networks within autonomous vehicle software," in *43rd International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2024.*

[16] Z. Chen, G. Li, and K. Pattabiraman, "A low-cost fault corrector for deep neural networks through range restriction," in *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*, pp. 1–13.

[17] A. Schmedding, L. Yang, A. Jog, and E. Smirni, "Aspis." [Online]. Available: https://doi.org/10.6084/m9.figshare.24757071.v1

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[19] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of CVPR'16*, 2016, p. 2818–2826.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *NeurIPS*, 2019.

[21] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR*, 2009.

[22] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," in *Computer Science Department, U. of Toronto*, 2009.

[23] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 97–108.

[24] L. Yang, B. Nie, A. Jog, and E. Smirni, "Practical resilience analysis of GPGPU applications in the presence of single- and multi-bit faults," *IEEE Trans. Computers*, vol. 70, no. 1, pp. 30–44, 2021.

[25] A. A. Khoshouyeh, F. Geissler, S. Qutub, M. Paulitsch, P. Nair, and K. Pattabiraman, "Structural coding: A low-cost scheme to protect cnns from large-granularity memory faults," in *Proceedings of SC 2023, Denver, CO, USA, November 12-17, 2023*.

[26] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault site pruning for practical reliability analysis of GPGPU applications," in *Proceedings of MICRO'18, Fukuoka, Japan, October 20-24, 2018*, pp. 749–761.

[27] L. Yang, B. Nie, A. Jog, and E. Smirni, "SUGAR: speeding up GPGPU application resilience estimation with input sizing," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 5, no. 1, pp. 01:1–01:29, 2021.

[28] A. Milluzzi and A. George, "Exploration of TMR fault masking with persistent threads on tegra gpu socs," in *2017 IEEE Aerospace Conference*. IEEE, 2017, pp. 1–7.

[29] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*, 2013, pp. 1310–1318.

[30] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren, "Patdnn: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning," *CoRR*, vol. abs/2001.00138, 2020.

[31] G. Li, S. K. S. Hari, M. B. Sullivan, T. Tsai, K. Pattabiraman, J. S. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proceedings of SC'17, Denver, CO, USA, November 12 - 17, 2017*.

[32] G. Kadam, E. Smirni, and A. Jog, "Data-centric reliability management in gpus," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 271–283.

[33] L. Yang, B. Nie, A. Jog, and E. Smirni, "Enabling software resilience in gpgpu applications via partial thread protection," in *Proceedings of ICSE'21*, 2021, pp. 1248–1259.

[34] J. Li, A. S. Rakin, Z. He, D. Fan, and C. Chakrabarti, "RADAR: run-time adversarial weight attack detection and accuracy recovery," in *Proceedings of DATE'21, Grenoble, France, February 1-5, 2021*, pp. 790–795.

[35] J. Ponader, K. Thomas, S. Kundu, and Y. Solihin, "MILR: mathematically induced layer recovery for plaintext space error correction of cnns," in *51st IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*, pp. 75–87.

[36] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," in *Proceedings of ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, pp. 397–408.

[37] J. Kim, M. B. Sullivan, and M. Erez, "Bamboo ECC: strong, safe, and flexible codes for reliable computer memory," in *Proceedings of HPCA'15, Burlingame, CA, USA, February 7-11, 2015*, pp. 101–112.

[38] S. Jha, S. Cui, T. Tsai, S. K. S. Hari, M. B. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "Exploiting temporal data diversity for detecting safety-critical faults in AV compute systems," in *52nd IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022, Baltimore, MD, USA, June 27-30, 2022*, pp. 88–100.

[39] A. Dosovitskiy, G. Ros, F. Codevilla, A. M. López, and V. Koltun, "CARLA: an open urban driving simulator," in *1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings*, ser. Proceedings of Machine Learning Research, vol. 78, pp. 1–16.

[40] H. Schafer, E. Santana, A. Haden, and R. Biasini, "A commute in data: The comma2k19 dataset," 2018.

[41] B. Zhou, Y. Sun, D. Bau, and A. Torralba, "Revisiting the importance of individual units in cnns via ablation," *CoRR*, vol. abs/1806.02891, 2018.

[42] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," *PLOS ONE*, vol. 10, pp. 1–46, 07 2015.

[43] X. Xie, T. Li, J. Wang, L. Ma, Q. Guo, F. Juefei-Xu, and Y. Liu, "NPC: neuron path coverage via characterizing decision logic of deep neural networks," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, pp. 47:1–47:27, 2022.

[44] S. Gerasimou, H. F. Eniser, A. Sen, and A. Çakan, "Importance-driven deep learning system testing," in *ICSE '20*, pp. 322–323.

[45] Y. Ibrahim, H. Wang, M. Bai, Z. Liu, J. Wang, Z. Yang, and Z. Chen, "Soft error resilience of deep residual networks for object recognition," *IEEE Access*, vol. 8, pp. 19 490–19 503, 2020.

[46] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," *Commun. ACM*, vol. 62, no. 11, p. 137–145, oct 2019.

[47] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of ICSE'18*, 2018, p. 303–314.

[48] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitras, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds., pp. 497–514.

[49] F. F. dos Santos, L. Carro, and P. Rech, "Kernel and layer vulnerability factor to evaluate object detection reliability in gpus," *IET Computers & Digital Techniques*, vol. 13, no. 3, pp. 178–186, 2018.

[50] C. Bolchini, L. Cassano, A. Miele, and A. Nazzari, "Selective hardening of cnns based on layer vulnerability estimation," in *2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2022, pp. 1–6.

[51] A. Mahmoud, S. K. Sastry Hari, C. W. Fletcher, S. V. Adve, C. Sakr, N. Shanbhag, P. Molchanov, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing selective protection for cnn resilience," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 2021, pp. 127–138.

[52] F. F. dos Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2018.

[53] K. Adam, I. I. Mohamed, and Y. Ibrahim, "A selective mitigation technique of soft errors for dnn models used in healthcare applications: Densenet201 case study," *IEEE Access*, vol. 9, pp. 65 803–65 823, 2021.