DeepDyve: Dynamic Verification for Deep Neural Networks

Yu Li*, Min Li*, Bo Luo, Ye Tian, and Qiang Xu
CUhk REliable Computing Laboratory (CURE)
Department of Computer Science and Engineering
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
Email: {yuli, mli, boluo, tianye, qxu}@cse.cuhk.edu.hk

ABSTRACT

Deep neural networks (DNNs) have become one of the enabling technologies in many safety-critical applications, e.g., autonomous driving and medical image analysis. DNN systems, however, suffer from various kinds of threats, such as adversarial example attacks and fault injection attacks. While there are many defense methods proposed against maliciously crafted inputs, solutions against faults presented in the DNN system itself (e.g., parameters and calculations) are far less explored. In this paper, we develop a novel lightweight fault-tolerant solution for DNN-based systems, namely **DeepDyve**, which employs pre-trained neural networks that are far simpler and smaller than the original DNN for dynamic verification. The key to enabling such lightweight checking is that the smaller neural network only needs to produce approximate results for the initial task without sacrificing fault coverage much. We develop efficient and effective architecture and task exploration techniques to achieve optimized risk/overhead trade-off in Deep-Dyve. Experimental results show that DeepDyve can reduce 90% of the risks at around 10% overhead.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; • **Hardware → System-level fault tolerance**.

KEYWORDS

Deep learning; Fault injection attack; Dynamic verification

1 INTRODUCTION

Machine learning with *deep neural networks* (DNNs) can produce results that have surpassed human-level performance in many challenging tasks lately, and it keeps improving. Consequently, DNNs

*These two authors contributed equally.

have become one of the foundation techniques in artificial intelligence (AI) applications. Lots of them (e.g., autonomous driving and medical image analysis) are safety- and security-sensitive.

Many researchers in the machine learning community believe that DNNs are rather robust to faults [10, 22], wherein removing some neurons or parameters leads to a graceful degradation in model accuracy. However, practical faults do not manifest themselves as the elimination of individual weights or neurons. Instead, they lead to bit-flips on DNN parameters or activations. Li et al. [23] conducted a case study on the Eyeriss DNN accelerator [4] under transient faults. Their results show that the FIT rate caused by random soft errors is far beyond the one required by the ISO 26262 standard (10 FIT[1]) [14] for the functional safety of road vehicles. Comparing to random errors, transient faults caused by malicious attacks are more severe. A recent attack named *DeepHammer* [44] shows that it can successfully tamper DNN inference behavior in practical setup, wherein the accuracy of multiple DNN classification systems are reduced to be as low as random guess within a few minutes, by leveraging the rowhammer vulnerability of DRAM used in the system.

There are a few recent research towards fault-tolerant DNN system designs. In [23], Li *et al.* proposed to set up a simple threshold to detect those faults that lead to drastic changes in DNN parameters. Such symptom-based error detectors have very little hardware overhead, but their detection capabilities are quite limited, especially for those quantized DNNs used in safety-critical embedded systems. They are also not applicable to defend against malicious faults [33, 44, 45]. To tackle this problem, a replication-based error detection technique for DNN systems was proposed in [24], but its overhead is quite high, requiring 40% extra computation to reach 60% fault coverage on CIFAR-10 dataset.

This paper aims for a solution with sufficient fault coverage yet little overhead for DNN-based classification systems. The term *dynamic verification* was introduced in [1], which detects errors in a complex super-scalar core by checking it with a core that is architecturally identical but micro-architecturally far simpler and smaller. Unlike symptom-based error detection techniques that check for abnormal behaviors, dynamic verification techniques check for in-variants in the system that are nearly always true over all possible executions. Following this idea, the proposed solution, namely *DeepDyve*, deploys a small neural network (referred to as the *checker DNN*) to approximate the original complex DNN model (referred to as the *task DNN*), and checks whether they produce *consistent* outputs in an end-to-end manner. If their results do not match, re-computation on the task DNN is performed for potential fault recovery, if any.

[1]Failure-in-Time Rate: 1 FIT = 1 failure per 1 billion hours.

Unlike [1], the result produced by the small checker DNN for error detection is only an approximate result. Hence, there will be both *false positives* (i.e., flagging nonexistent failures) and *false negatives* (i.e., miss to report failures) with DeepDyve. False positives result in unnecessary re-computation cost, while false negatives lead to fault coverage loss. Consequently, it is essential to achieve high *consistency* between the task DNN and the checker DNN under the fault-free situation. We formulate the checker DNN design problem as a design exploration problem, wherein we evaluate a set of candidate small DNN designs for the given big task DNN model and choose the one with optimized coverage/overhead trade-off as the checker DNN.

On the one hand, we obtain the set of candidate checker DNNs by compressing and transferring the task DNN model's knowledge through knowledge distillation [11]. On the other hand, the given classification task might be too complicated for the much smaller checker DNNs. Under such circumstances, we allow the checker DNN to perform the classification task with reduced complexity. For example, given a task to classify ten objects wherein two kinds of objects are easily confused, we could allow the checker DNN to perform a simpler problem with these two objects treated as one class. By doing so, there will be much less false positives in dynamic verification, at the cost of more false negatives since we would not be able to identify those faults that result in misclsssification between these two kinds of objects. Consequently, we need to carefully perform task simplification for the checker DNN design to strike an optimized balance between coverage and overhead.

From the safety and security perspective, misclassifying different classes often has quite different impacts. For example, in a traffic sign recognition system, misclassifying a "Yield" sign as a "Stop" sign does not cause much trouble, but the opposite misclassification may cause severe traffic accidents. We need to consider such risk impacts in the checker DNN design. That is, we should be more concerned about the risk/overhead trade-off instead of coverage/overhead trade-off.

The proposed DeepDyve solution considers the above issues, and the main contributions of this paper include:

- To the best of our knowledge, DeepDyve is the first dynamic verification technique for resilient DNN designs, which uses a far simpler and smaller checker DNN for online error detection and recovery.
- We propose a novel two-stage checker DNN design methodology, which explores both the checker DNN architectures and task simplification possibilities. In particular, we propose a novel checker DNN architecture exploration technique with theoretical guarantees. Also, being able to manipulate the tasks performed on the checker DNN dramatically increases the solution space of DeepDyve.
- DeepDyve leverages the uneven risk probabilities and safety impact among classes to guide the design exploration procedure. Experimental results on CIFAR-10, GTSRB, CIFAR-100, and Tiny-ImageNet datasets show that it can reduce up to 90% of the risks at around 10% computational overhead.

We organize the remainder of this paper as follows. Section 2 presents the preliminaries and motivation of this work. Then, we give an overview of DeepDyve in Section 3. Next, we detail the checker DNN architecture exploration and task exploration techniques in Section 4 and 5, respectively. Experimental results are presented in Section 6. Then, we discuss the applications and limitations of DeepDyve in Section 7, followed by the survey of related works in Section 8. Finally, Section 9 concludes this paper.

2 PRELIMINARIES AND MOTIVATION

In this section, we first present the impact of hardware faults (random or malicious) on the reliability and security of DNN systems in Section 2.1. Then, we provide the threat model in Section 2.2. At last, we illustrate the motivation for the proposed DeepDyve solution in Section 2.3.

2.1 DNN Systems under Faults

There are mainly two types of attacks during the inference of DNN systems: adversarial example attack and fault injection attack. Adversarial example attacks [9, 28] try to fool DNN systems by crafting subtle malicious perturbations on inputs, and there is a vast body of research on this topic [20, 27, 31, 32]. By contrast, fault injection attack aims to break the system by injecting faults into the internal system execution pipeline, e.g., flipping data bits in processing elements. This problem is less explored in the literature, and the overhead of existing defense techniques is quite high.

Various types of transient faults can be introduced into DNN systems by attackers, e.g., clock glitch attack [30], voltage glitch attack [36], and rowhammer attack [17]. Besides malicious faults, transients faults could also occur due to environmental perturbations such as alpha particle strikes.

The Impact of Transient Faults. Transient faults may occur at the data paths and buffers of processing units [23] or inside the memories of DNN-based systems [12]. Such faults would manifest themselves as errors in DNN calculations or intermediate values [34] during inference. A failure occurs when errors propagate to the outputs of the system and cause behavioral changes, which could lead to catastrophic consequences in safety-critical applications, e.g., misclassifying a "Stop" sign as a "Yield" sign in autonomous vehicles and taking the wrong action. In the following, if not specified, faults, errors and failures are used in an exchangeable manner, and we are only concerned with those faults that cause misclassifications.

Existing Attacks. Recently, DNNs are shown to be vulnerable to fault injection attacks [26]. In [12], the authors estimated that 40–50% of the parameters in a DNN model could lead to an accuracy drop greater than 10% when bit-flips occur in their data representation. While it was shown that quantized DNNs are more resilient to fault injection attacks, the recent progressive BitFlip Attack (BFA) [33] proposed by Rakin et. al. can reduce the accuracy of a quantized ResNet-18 from 68.9% to 0.1% with only 13 bit-flips. BFA combines gradient ranking and progressive search to identify those vulnerable bits that degrade model accuracy significantly when flipped. The follow-up work DeepHammer by Yao *el al.* [44] proves the effectiveness of BFAs in practice, by row-hammering against various real DNN systems.

Existing Defenses. For DNNs implemented on floating-point machines, only a small fraction of the dynamic range provided by
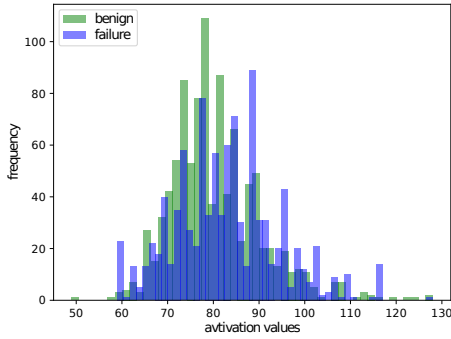
Figure 1: Comparison of the maximum activation value between benign and failure cases (because of fault attack) from a quantized (8-bit integer) classifier.

Table 1: A Motivational example.

Checker DNN	Size	Accuracy	Computational Overhead
A	1%	80%	∼21%
B	5%	92%	∼13%
C	10%	94%	∼16%

launch random bit-flips, while for strong attackers, they can precisely locate and launch fault injection in the processing pipeline.

Moreover, we assume the same transient faults would not occur in consecutive DNN inference runs. Firstly, reliability threats rarely occur and the probability to occur repeatedly in a short period is negligible. Secondly, it is very difficult, if not impossible, to launch the same faults repeatedly in a DNN system. For example, in [44], launching a rowhammer attack requires long preparation time (several minutes). As long as the DNN inference time is short (and usually it is), attackers do not have sufficient time to launch the same attack in the second run.

2.3 Motivation

For any classification problem, there could be many DNN models with different size/accuracy trade-offs to solve it. Although big models often have higher accuracy, many inputs can be correctly handled by small models. Therefore, we could employ a smaller checker DNN to perform the same task, and they should output the same results in *most* cases when faults do not occur. In this way, the task model can be dynamically verified for online error detection and recovery. Note that, it is not possible to achieve deterministic dynamic verification for such systems because the outputs of a simple model cannot achieve 100% consistency with that of the original model.

As discussed earlier, because the checker DNN is less accurate, there will be false positives and false negatives. Generally speaking, the larger the checker DNN is, the more accurate it is [40], but it does not necessarily lead to larger computational overhead. We use the following example to illustrate the impact of checker DNN design.

Suppose the task DNN model is with ∼100% accuracy, and there are three candidate checker DNN models: A, B, and C. Their relative sizes compared to the task DNN and their classification accuracy are shown in the second and the third column of Table 1. Consider transient fault-induced failures are rare events, the computational overhead of the three models is estimated in the fourth column, which is the sum of the computational cost of the checker DNN itself (as it is always on) and the re-computation cost on the task DNN when the checker DNN produces a different classification result (false positive cases).

If the misclassifications caused by faults are evenly distributed among all classes, the fault coverage would be similar to the accuracy of the checker DNN. For this particular example, checker B can achieve ∼92% fault coverage with 5% hardware cost and ∼13% computational cost, which significantly outperforms existing fault-tolerant solutions for DNN designs. This has motivated the proposed DeepDyve solution in this paper.

In practice, fault-induced misclassifications are usually not evenly distributed. Moreover, misclassifying different classes often have

the data type is used. If a fault makes the magnitude of intermediate output values huge, it is likely to lead to a failure. Based on this observation, Li *et al.* proposed to use a simple threshold to detect those faults that lead to intermediate outputs beyond it [23]. To be specific, before deployment, they record the value ranges (X_{min}, X_{max}) of the output for each layer. After deployment, the output value range is then checked in the run-time. They consider a fault is detected if there are output values beyond the range $(1.1 \times X_{min}, 1.1 \times X_{max})$. This anomaly detector has very little hardware overhead, but its detection capabilities are quite limited, especially for recent attacks on quantized DNN systems. We show an example in Figure 1 where the threshold-based anomaly detector fails to detect faults in quantized classifiers. For the threshold detector to detect faults, at least the maximum activation value should be beyond the normal range. However, our experiment shows that the maximum activation values from all failure cases (due to hardware fault) are not bigger than the normal boundary.

Li *et al.* [24] design a replication-based error detection technique for deep neural networks. However, their overhead is quite high. They spend 40% overhead to reach 60% fault coverage on CIFAR-10. In the DeepHammer work [44], the authors discuss a few potential mitigation techniques but do not provide any quantitative results for their effectiveness.

2.2 Threat Model

In this work, we consider the attacker is trying to compromise the accuracy of a DNN system by maliciously injecting faults into it. Unlike crafting malicious inputs to fool DNN systems, we target faults presented in the system internals, i.e., processing elements, buffered weights, and intermediate values stored in on-chip buffers or memories, and so on. We consider the attacker succeeds if the model's output class is different from the one obtained in an attack-free environment.

We assume the attackers have full knowledge of the DNN and its deployment on the device, including neural network topology, parameters, and low-level implementation details, e.g., the position of intermediate values stored in memories. Weak attackers could

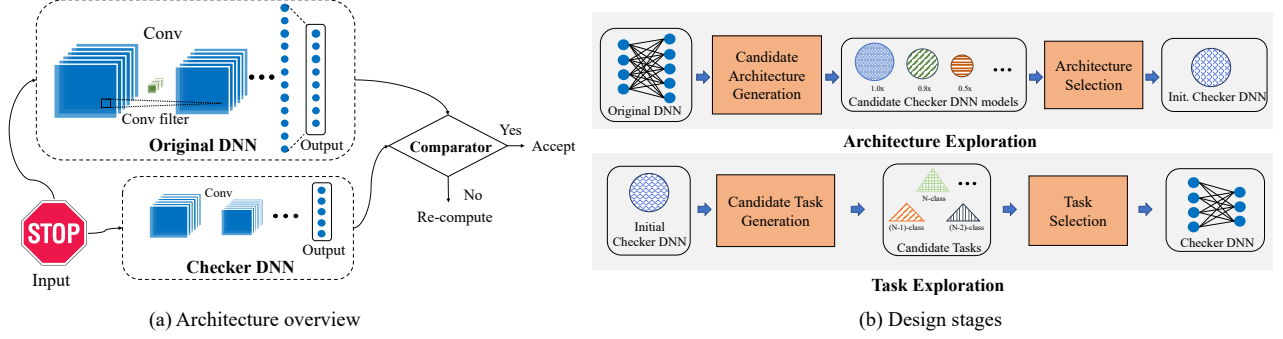(a) Architecture overview   (b) Design stages

Figure 2: DeepDyve architecture and the design flow.

different risk implications for safety-critical systems. Therefore, for any possible checker DNN, we need to consider the risk impact and apply fault simulation to evaluate its effectiveness. This is a time-consuming process. Therefore, the critical challenge is how to efficiently explore the solution space of all possible checker DNN designs to find the one with an optimized risk/overhead trade-off.

3 DEEPDYVE OVERVIEW

Figure 2 (a) depicts the proposed DeepDyve architecture. It contains three parts: the original network, the checker DNN, and the comparator. The checker DNN is a smaller and simpler DNN model which approximates the original network and the original task. An input instance is processed by both of the two models. Their outputs are checked by the comparator and accepted if consistent. Otherwise, the input instance is subject to a re-computation by the original model, and the new prediction is accepted, regardless of whether the two model outputs are consistent or not.

Design Goals. We first formally define the evaluation metrics (e.g. coverage and overhead). We evaluate the cost of checker DNN design by the introduced overhead. The overhead of the checker DNNs can be calculated as follows:

$$O(S) = \frac{params_{small}}{params_{big}}, \tag{1}$$

$$O(C) = \frac{FLOP(net_{small}) + (1 - P_{consistent}) \times FLOP(net_{big})}{FLOP(net_{big})}, \tag{2}$$

$O(S)$ and $O(C)$ stand for the storage overhead and computational overhead, respectively, wherein *params* stands for the storage requirement of model parameters with unit of Mega Bytes (*MB*), and *FLOP* function calculates the number of multiply-accumulation operations in the network. The computational overhead contains two parts: FLOP of the small network (static overhead) and the re-computation overhead (dynamic overhead) when the small network output is different from that of the big one (with probability $1 - P_{consistent}$).

The detection ability of DeepDyve is characterized by the coverage rate. A popular definition of fault coverage would be number of classification failures detected among all mis-classifications caused by faults, as show in in Equation 3. $DF_{i,j}$ stands for the detected failures mis-classified from class i to class j, and $TF_{i,j}$ denotes the total failures from class i to class j when faults occur. To take the

different risk impact of different failures on safety-critical application into consideration, we introduce a new metric called *weighted coverage*, abbreviated as $WCov$. in Equation 4. $I_{i,j}$ is the risk impact if class i is mis-classified into class j, which will be defined later in this Section. Note that Cov. is a special case of $WCov$. when all misclassifications have the same risk impact. In later text, we use coverage and weighted coverage interchangeably and they both refer to weighted coverage if not specified.

$$Cov. = \frac{\sum_{ij} DF_{ij}}{\sum_{ij} TF_{ij}}. \tag{3}$$

$$WCov. = \frac{\sum_{ij} DF_{ij} \times I_{ij}}{\sum_{ij} TF_{ij} \times I_{ij}}, \forall i, j \in N \text{ and } i \neq j. \tag{4}$$

Design Stages. Under the guidance of the design goals, there are mainly two stages in designing of the checker DNN and we show them in Figure 2 (b). The first stage is architecture exploration, where we initialize the architecture of the checker DNN. Given a task model, a pool of checker DNN candidates with the same task of the given model are generated with model compression techniques. Then, one of them is picked from the pool by evaluating their overhead and fault coverage, detailed in Section 4. The second stage is task exploration, where we try to manipulate the classification tasks performed by the checker DNN to achieve better coverage/overhead trade-off. That is, we can find a better solution by providing more design options at the task level, detailed in Section 5.

To solve the above design exploration problems, we define the following three matrices:

- *Risk impact matrix* $\mathbf{I} \in \mathbb{R}^{N \times N}$. In safety-critical DNN applications, the risk impact of different misclassifications may vary significantly from the system perspective. Each entry in $\mathbf{I}$ denotes the cost of the corresponding misclassification (the larger the value, the higher the cost). As the actual risk impact depends on the application, the values in the impact matrix should be carefully determined by system designers.
- *Risk probability matrix* $\mathbf{R} \in \mathbb{R}^{N \times N}$, where the entry R_{ij} denotes the probability that the i-th class is misclassified to j-th class when faults occur, and N denotes the total number of classes. Risk probability matrix is obtained from fault injection experiments. Figure 3 shows an example of $\mathbf{R}$ drawn from CIFAR-10 dataset by performing random fault injection on VGG-16 for 400,000 times. From this example, we can
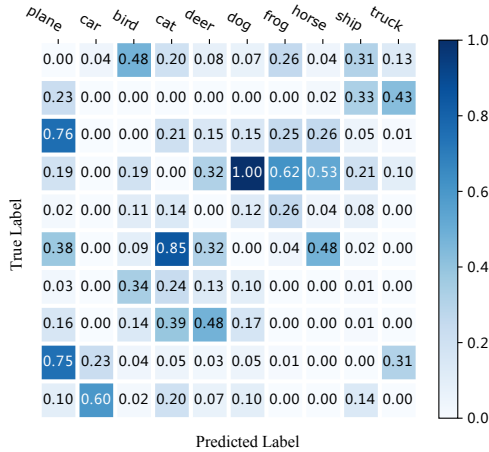
Figure 3: An example of risk probability matrix obtained from CIFAR-10 (normalized for visualization).

observe fault-induced misclassifications are far from evenly distributed.

- *Inconsistency matrix* $\mathbf{C} \in \mathbb{R}^{N \times N}$, wherein each entry C_{ij} denotes the probability that one sample is labeled as i-th class by the task model while is labeled as j-th class by the small model in DeepDyve. Note that entry $C_{ii}, i \in [1..N]$ equals to zero. In practice, some classes are naturally more difficult to classify (e.g., *dog* and *cat* in the CIFAR-10 dataset) than others. These difficult classes cause more inconsistency than the easy ones. Combining them as one class (e.g., as *pet*) in the checker DNN is relatively easy to achieve high consistency.

Risk impact matrix $\mathbf{I}$ will be used in calculating weighted coverage. Besides, the three matrices $\mathbf{I}$, $\mathbf{R}$ and $\mathbf{C}$ will be all used in task exploration, wherein we try to combine those classes that are easily confused yet have less risk for task simplification.

4 ARCHITECTURE EXPLORATION

The objective of the architecture exploration procedure is to find a initial checker DNN model that achieves good fault coverage with low overhead defined in Section 3. To this end, firstly, we generate a pool of checker DNN candidates. The generation process is trying to minimize the overhead with the help of model compression techniques proposed in [13], detailed in Section 4.1. Second, as different candidates offer different trade-offs between overheads and coverage, we illustrate how to efficiently search for an appropriate checker DNN design from the candidates in Section 4.2.

4.1 Checker DNN Candidate Generation

In DeepDyve, the *consistency* between predictions of the checker DNN and those of the original DNN decides the computational overhead $O(C)$. Consider an input that is mis-classified by the original DNN when no faults occur, we would like to have the checker DNN output the same wrong label, so that DeepDyve does not flag a nonexistent failure, avoiding unnecessary re-computation.

To improve consistency, given the task DNN model, we use model compression to generate the checker model candidates. Specifically,

Table 2: ResNet-10 with different width multiplier.

α	Accuracy(%)	O(S) (MB)	O(C) (GFLOPs)
1.0	97.54	1.23	0.06
0.7	96.94	0.60	0.03
0.5	96.20	0.31	0.02
0.3	95.75	0.12	0.01

we use two types of model compression techniques. First, we use architecture compression to search for the potential architectures and then we use knowledge distillation to train our checker DNN.

Architecture Compression. No doubt to say, the amount of available design choices has a significant impact on any design exploration problem. In order to increase the design options for DeepDyve, we adopt the model compression approach in [13] to make the size of checker DNNs adjustable. To be specific, given the task DNN architecture, we use a single width multiplier α to adjust it, by uniformly scaling down the number of channels (or neurons if it is a linear layer) for each layer. For example, a feature map with 100 channels will be scaled down to the one with ten channels with α being set to 0.1. By applying width multiplier, the resulting model architecture has much less overhead.

We take one of popular architectures—ResNet trained on GT-STB [39] as a case study to show the effect of width multiplier. Table 2 lists the accuracy, the storage overhead (in MegaByte) and the computational overhead (in Giga Floating Point Operations) of ResNet-10 with different width multipliers. The first row stands for the original ResNet-10. As can be observed, accuracy drops smoothly with smaller model size and less computational cost.

Parameter Training. To further improve the consistency between the task DNN and the checker DNN, we use *knowledge distillation* to train the checker DNN. Knowledge distillation, formulated by Hinton *et al.* [11], is a training solution to distill a task model (teacher model) and transfer knowledge to a simpler model (student model).

In our training, the first step of knowledge distillation from the task DNN is to covert the pre-softmax logits, z_i, computed for each class into a probability, p_i, by Equation 5 with the *temperature T*.

$$p_i = \frac{exp(z_i/T)}{\sum_j exp(z_j/T)} \quad (5)$$

With higher temperature, the new targets for the checker DNN to learn are 'softer' probability distributions over classes.

Next, the checker DNN is trained by minimizing the knowledge distillation loss (L_{KD}), which is defined as:

$$L_{KD} = \lambda T^2 \times CrossEntropy(P_C^T, P_O^T)$$
$$+ (1 - \lambda) \times CrossEntropy(P_C, y_{true}), \quad (6)$$

wherein P_C^T and P_O^T are the softened outputs of the checker DNN and the original DNN under the same temperature T. The first component of L_{KD} forces the checker DNN towards approximating similar output distribution of the original DNN (i.e., consistency), whereas the second component of L_{KD} forces the checker DNN towards correctly classifying inputs as usual (i.e., accuracy). We use λ to tune the weighted average between the kinds of losses.
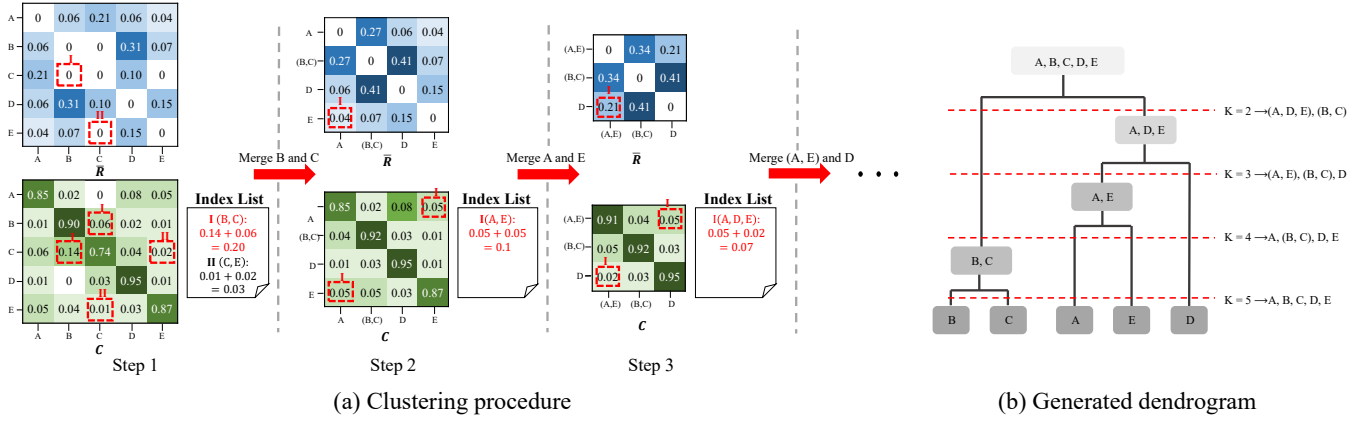
Figure 4: An illustrative example of agglomerative class clustering.

4.2 Search Strategy

Our search strategy is based on the empirical observation that the consistency between the two models in DeepDyve is related to the multiplier α used to generate the small checker DNN ($\alpha << 1$). Here we formally define the *consistency function* of α by:

DEFINITION 1. *Let $g_t(\mathbf{x})$ and $g_\alpha(\mathbf{x})$ be the task DNN model and the checker DNN model generated from the task DNN with a multiplier α. Each model g takes a vector of input $\mathbf{x}$ and outputs a class of $y \in (0..N-1)$, where N is the number of classes. We define the consistency function as the probability that the outputs are consistent over the input space X.*

$$f(\alpha) = P(g_t(\mathbf{x}) = g_\alpha(\mathbf{x})), \mathbf{x} \in X \tag{7}$$

We choose a simple function with the form of $f(\alpha) = -\frac{a}{\alpha} + b$ to approximate the consistency function. In this function, a and b are positive parameters whose values can be obtained via curve fitting after collecting a number of checker DNN models with different α and the corresponding consistency values. Besides, because $0 \le f(\alpha) \le 1$ and $0 < \alpha \le 1$, we can obtain the valid range of α, which is $\frac{a}{b} \le \alpha \le 1$.

Once a and b are known, we have the following theorem:

THEOREM 4.1. *Suppose $f(\alpha)$ is approximated with $-\frac{a}{\alpha} + b$, when $a > 0, b > 0$, and $\frac{a}{b} \le \alpha \le 1$. We can find an optimal $\alpha = \sqrt[3]{\frac{a}{2}}$ where the computational overhead $O(C)$ is minimized.*

PROOF. First, for a neural network composed of linear and convolutional layers, the FLOPs of the checker DNN with a multiplier α is α^2 times the original FLOPs. Recall that the number of floating point operations (FLOPs) for one linear layer can be estimated by:

$$2 \times I \times O \tag{8}$$

where I and O are number of input and output neurons in one linear layer, respectively. Therefore, the FLOP of a compressed linear layer with multiplier α is:

$$\alpha^2 \times (2 \times I \times O) \tag{9}$$

Similarly, for a convolutional layer, the floating point operations with multiplier α is estimated by:

$$(2 \times k^2 \times C_{in}) \times (H_{out} \times W_{out} \times C_{out}) \tag{10}$$

where k stands for the kernel size, and C_{in}, C_{out} stands for number of input and output channels, respectively. H_{out} and W_{out} are the height and the width of the output tensors. Given this, the FLOP of a compressed convolutional layer with multiplier α is:

$$\alpha^2 \times (2 \times k^2 \times C_{in}) \times (H_{out} \times W_{out} \times C_{out}). \tag{11}$$

Hence, if we add all layers together, the final FLOPs of the checker DNN will be α^2 times of the task model where $\alpha = 1$.

Providing this, the computational overhead of DeepDyve with the checker DNN can be simplified from Equation 2 to Equation 12.

$$O(C) = \alpha^2 + (1 - f(\alpha)), \alpha \in (0, 1] \tag{12}$$

$$O(C) = \alpha^2 + (1 + \frac{a}{\alpha} - b), \alpha \in (0, 1], a > 0, b > 0, \tag{13}$$

$$\nabla O(C) = 2\alpha - \frac{a}{\alpha^2}, \alpha \in (0, 1], a > 0 \tag{14}$$

$$\text{Let } \nabla O(C) = 0, \text{ then } \alpha = \sqrt[3]{\frac{a}{2}} \tag{15}$$

To find the optimal point, we calculate the gradient of $O(C)$ as Equation 14. By letting the gradient equals to 0, we obtain the optimal point of α, which is $\sqrt[3]{\frac{a}{2}}$, as shown in Equation 15. □

Therefore, to obtain the optimal α, we are going to fit the consistency function $-\frac{a}{\alpha} + b$ with the given candidate pool. After that, we select the checker DNN with $\alpha = \sqrt[3]{\frac{a}{2}}$.

5 TASK EXPLORATION

After the initial checker architecture is fixed, DeepDyve performs task exploration to achieve better risk/overhead trade-off. In Section 5.1, we first discuss how to perform task simplification efficiently under the guidance of risk probability matrix $\mathbf{R}$, risk impact matrix $\mathbf{I}$ and inconsistency matrix $\mathbf{C}$. The first step generates a

bunch of different tasks. Then, we detail the search strategy to select the best task in Section 5.2.

Algorithm 1: Agglomerative class clustering

Input: Risk matrix $\overline{\mathbf{R}}$, inconsistency matrix $\mathbf{C}$, No. of classes N, class labels L

Output: $(N - 2)$ cluster label lists

  /* Initialize   */

1   $k = N - 1$;

2   $candidateList = []$;

3   **for** $q = 1, 2, \ldots, N$ **do**

4 | $G_q = l_q$;   // Cluster with single class

5 | $\lambda_q = q$;   // Initialize cluster label

6   **end**

7   **while** $k \geq 2$ **do**

       /* Select clusters based on two criteria   */

8 | $indexList = $ all $\arg\min_{i,j} COV_{loss}(i, j)$ in $\overline{\mathbf{R}}$;

9 | $(n, m) = \arg\max_{i,j} O_{save}(i, j)$ in $indexList$;

       /* Update cluster label list, matrices   */

10 | Merge G_n and G_m, Update clusters $\{G\}$ and $\boldsymbol{\lambda}$;

11 | Update $\overline{\mathbf{R}}$, $\mathbf{C}$;

       /* Add cluster label list to candidates   */

12 | $candidateList[k] = \boldsymbol{\lambda}$;

13 | $k = k - 1$;

14   **end**

15   **return** $candidateList$;

5.1 Agglomerative Class Clustering

Given the original N-class task, our problem is to find a simplified K-class task for any given checker DNN with better overhead/coverage trade-off. We consider it as a clustering problem and propose the *Agglomerative Class Clustering* to solve it (see Algorithm 1).

Formally, we assume the labels of original N classes as: $L = \{l_1, l_2, \ldots, l_N\}$. For the sake of simplicity, we can map the labels into integer numbers, as $L = \{1, 2, \ldots, N\}$. They are to be grouped into K clusters $\{G_k | k = 1, 2, \ldots, K\}$, where $G_{k'} \bigcap_{k' \neq k} G_k = \varnothing$ and $L = \bigcup_{k=1}^{K} G_k$. Accordingly, we can use $\lambda_i \in \{1, 2, \cdots, K\}$ to represent the cluster label of original label l_i. Then, the clustering result can be represented by a cluster label list: $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \cdots, \lambda_N)$.

Risk matrix $\overline{\mathbf{R}}$. The risk probability matrix and risk impact matrix can be integrated into one single risk matrix with an element-wise multiplication:

$$\overline{\mathbf{R}} = \mathbf{R} \odot \mathbf{I}, \tag{16}$$

wherein each entry in $\overline{\mathbf{R}}$ stands for the risk between classes of the big network.

Two clustering criteria. Merging two classes into one cluster have two effects:

- Coverage loss, since fault-induced misclassifications between the two classes cannot be detected any more. Hence, we prefer merging classes with small values in risk matrix $\overline{\mathbf{R}}$.

- Overhead savings, because the simplified task is easy to learn and it will be more consistent with the big DNN, thereby reducing re-computational overhead.

We use COV_{loss} and O_{save} to denote such effects, which are used in Algorithm 1.

Updating $\overline{\mathbf{R}}$ and $\mathbf{C}$. After selecting two classes or clusters to merge, we should update the $\overline{\mathbf{R}}$ and $\mathbf{C}$ accordingly. Assuming cluster G_i and G_j ($i < j$) are to be merged, we first move classes in cluster G_j to G_i, as $G_i = G_i \bigcup G_j$, delete G_j and re-assign the cluster label for the rest of clusters. Then we update the risk and inconsistency values of G_i (i-th row and i-th column of $\overline{\mathbf{R}}$ and $\mathbf{C}$) as the sum of the corresponding values of two clusters. Lastly, we delete the j-th row and j-th column in $\overline{\mathbf{R}}$ and $\mathbf{C}$. In this way, we aggregate the risk probability and inconsistency values of two merged clusters while preserving the property of the matrices defined in Section 3.

Clustering scheme. We apply a hierarchical clustering algorithm—*agglomerative class clustering*, which is illustrated in Algorithm 1. Specifically, each class of the original task starts in its own cluster. Then, we search for two clusters with the smallest coverage loss in $\overline{\mathbf{R}}$. Note that $\overline{\mathbf{R}}$ (see Figure 3) is usually sparse, in case of multiple occurrences of the minimum value, we choose the one with the largest overhead savings from $\mathbf{C}$, which in turn improves the model consistency between the task model and the checker DNN. Next, we merge the two selected clusters into one cluster and update $\overline{\mathbf{R}}$ and $\mathbf{C}$. The above procedure iterates in a bottom-up manner until all classes of the original task are merged as a single cluster. Consequently, the clustering results can be presented in a dendrogram with $K = 2$ to $K = N - 1$ clustering candidates, which enables later exploration for the optimal simplified task for a given checker DNN. Figure 4 shows an example of the iterative clustering procedure with five classes and the generated dendrogram.

5.2 Search Strategy

After obtaining candidate tasks from Algorithm 1, we evaluate the corresponding overhead savings and coverage loss with fault injection experiments. As the number of candidate tasks $N - 1$ increases linearly with the number of original classes, we can evaluate them efficiently. Afterwards, we have a list of pareto-optimal checker DNN designs with various coverage/overhead trade-offs. We then choose the optimal final DNN design.

6 EXPERIMENTAL RESULTS

In this section, we demonstrate the effectiveness of DeepDyve. First, we present our experimental setup in Section 6.1. After that, our results that architecture exploration facilitates to find an optimized checker DNN architecture in Section 6.2, and task simplification further saves the overhead of DeepDyve in Section 6.3, respectively. We show DeepDyve outperforms existing solutions in Section 6.4. At last, in Section 6.5, we discuss the impact of model accuracy through a case study on CIFAR-100.

6.1 Setup

Datasets. We demonstrate the effectiveness of DeepDyve on four widely used image classification datasets: CIFAR-10 [19], The German Traffic Sign Recognition Benchmark (GTSRB) [39], CIFAR-100 [19], and Tiny-ImageNet [21]. CIFAR-10 and CIFAR-100 datasets
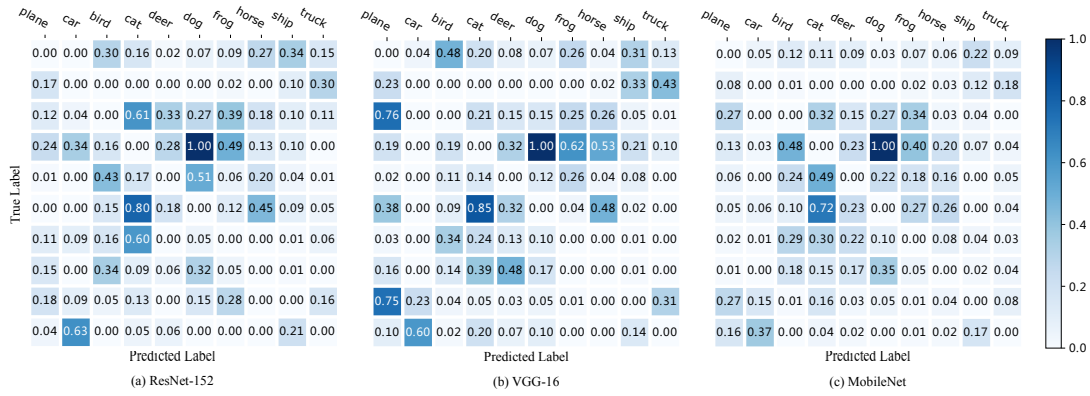
(a) ResNet-152

True Label	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	0.00	0.00	0.30	0.16	0.02	0.07	0.09	0.27	0.34	0.15
car	0.17	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.10	0.30
bird	0.12	0.04	0.00	0.61	0.33	0.27	0.39	0.18	0.10	0.11
cat	0.24	0.34	0.16	0.00	0.28	1.00	0.49	0.13	0.10	0.00
deer	0.01	0.00	0.43	0.17	0.00	0.51	0.06	0.20	0.04	0.01
dog	0.00	0.00	0.15	0.80	0.18	0.00	0.12	0.45	0.09	0.05
frog	0.11	0.09	0.16	0.60	0.00	0.05	0.00	0.00	0.01	0.06
horse	0.15	0.00	0.34	0.09	0.06	0.32	0.05	0.00	0.01	0.00
ship	0.18	0.09	0.05	0.13	0.00	0.15	0.28	0.00	0.00	0.16
truck	0.04	0.63	0.00	0.05	0.06	0.00	0.00	0.00	0.21	0.00

Predicted Label

(b) VGG-16

True Label	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	0.00	0.04	0.48	0.20	0.08	0.07	0.26	0.04	0.31	0.13
car	0.23	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.33	0.43
bird	0.76	0.00	0.00	0.21	0.15	0.15	0.25	0.26	0.05	0.01
cat	0.19	0.00	0.19	0.00	0.32	1.00	0.62	0.53	0.21	0.10
deer	0.02	0.00	0.11	0.14	0.00	0.12	0.26	0.04	0.08	0.00
dog	0.38	0.00	0.09	0.85	0.32	0.00	0.48	0.00	0.02	0.00
frog	0.03	0.00	0.34	0.24	0.13	0.10	0.00	0.00	0.04	0.03
horse	0.16	0.00	0.14	0.39	0.48	0.17	0.00	0.00	0.01	0.00
ship	0.75	0.23	0.04	0.05	0.03	0.05	0.01	0.00	0.00	0.31
truck	0.10	0.60	0.02	0.20	0.07	0.10	0.00	0.00	0.14	0.00

Predicted Label

(c) MobileNet

True Label	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	0.00	0.05	0.12	0.11	0.09	0.03	0.07	0.06	0.22	0.09
car	0.08	0.00	0.01	0.00	0.00	0.00	0.02	0.03	0.12	0.18
bird	0.27	0.00	0.00	0.32	0.15	0.27	0.34	0.03	0.04	0.00
cat	0.13	0.03	0.48	0.00	0.23	1.00	0.40	0.20	0.07	0.04
deer	0.06	0.00	0.24	0.49	0.00	0.22	0.18	0.16	0.00	0.05
dog	0.05	0.06	0.10	0.72	0.23	0.00	0.27	0.26	0.00	0.04
frog	0.02	0.01	0.29	0.30	0.22	0.10	0.00	0.08	0.04	0.03
horse	0.01	0.00	0.18	0.17	0.35	0.05	0.00	0.00	0.02	0.04
ship	0.27	0.15	0.01	0.16	0.03	0.05	0.01	0.04	0.00	0.08
truck	0.16	0.37	0.00	0.04	0.02	0.00	0.01	0.02	0.17	0.00

Predicted Label

Figure 5: Risk probability matrices of different models under four million fault injections.

Table 3: Datasets and Task Models.

Dataset	#Classes	Task Model	Accuracy	FLOPs	Parameter
CIFAR-10	10	ResNet-152	96.15%	3.75 G	58.22 M
GTSRB	43	ResNet-34	99.40%	1.16 G	21.30 M
CIFAR-100	100	ResNet-152	80.11%	3.75 G	58.22 M
Tiny-ImageNet	200	WideResNet-101	85.20%	22.84 G	126.89 M

contain 50,000 training images and 10,000 test images, and they have 10 and 100 classes, respectively. GTSRB has 43 classes of different traffic signs. It has 39,209 training images and 12,630 test images in total. Tiny-ImageNet is a 200-class natural image dataset sub-sampled from ImageNet dataset and it contains 100,000 training and 10,000 validation images.

Models. Table 3 shows the task DNN used by each dataset. For CIFAR-10, the task DNN model is a ResNet-152 with an accuracy of 95.16%. For GTSRB, the task DNN we use is a ResNet-34 model with an accuracy of 98.6%. The ResNet-152 for CIFAR-100 has an accuracy of 80.11%. The task model for Tiny-ImageNet is WideResNet-101, and its accuracy is 85.20%. Please note that for Tiny-ImageNet, we use pre-trained weights on the ImageNet dataset and fine-tune them to obtain high accuracy. We use Pytorch profiling tool "thop"[2] to quantify model GFLOPs and parameters. We quantize all DNN parameters into 8-bit integers (INT-8) following a uniform affine quantitizer [18].

Fault Model. In our experiment, we use two types of fault injection: random fault injection and BitFlip Attack (BFA) [33]. For random fault injection, in each simulation run, we randomly flip n bits in the model and pass one randomly selected image to the DNN model for inference. BFA proposed in [33] is the state-of-art fault injection attack on DNN models. It can crash the DNN system by injecting a small number of bit-flips by searching the most vulnerable bit progressively. A failure occurs when the predicted label is different from the one obtained in the fault-free case.

Risk Impact Martrix. In practice, the risk impact matrix should be determined by system designers after conducting application-specific risk analysis. One practical solution would be categorizing

the risk impact into a few risk levels and filing the matrix accordingly. In our experiments. we simulate the impact matrix with two configurations.

- **Uniform Impact**, where all entries are ones. It represents that the risk impact among all classes are equal. When the risks of different classes do not have significant differences, the uniform impact matrix can be used for simplicity.
- **Non-uniform Impact**, where the risk impact values are set to two different levels. As classes with the low precision are not trustworthy themselves and hence have low risk, in this configuration, we assign those classes with the lowest 25% precision with impact 1 and the others with 100.

Risk Probability Matrix. We obtain the risk probability matrix and the failure coverage of the checker DNN with random fault injection experiments. Figure 5 shows the risk probability matrices of different state-of-art model architectures trained on CIFAR-10. We perform 4 million fault injections to obtain the result in this figure[3]. For visualization purposes, the probability matrix is divided by the maximum element. The sum of all elements in the original probability matrix is 1.

We can observe that the risk probability matrix is more task-related than model-related. The probability distributions are very similar across different DNN models on the same CIFAR-10 task. For example, in Figure 5 (a), the value between dog and cat is the highest one with fault injections. It is also true for Figure 5 (b) and (c). This matrix will be used in the task simplification process.

6.2 Effectiveness of Architecture Exploration

In this part, we study the effectiveness of the architecture exploration.

Figure 6 shows the consistency, indicated by blue points and approximated by blue curve, and the computational overhead, indicated by red points and connected by the red curve, of checker DNN models with different sizes trained by DeepDyve. Please note that we only investigate five checker model sizes for Tiny-ImageNet due to high training effort for this dataset, including the pre-training on ImageNet dataset and fine-tuning on Tiny-Imagenet. For all the four datasets, the consistency between the task and checker models

---

[2]https://github.com/Lyken17/pytorch-OpCounter

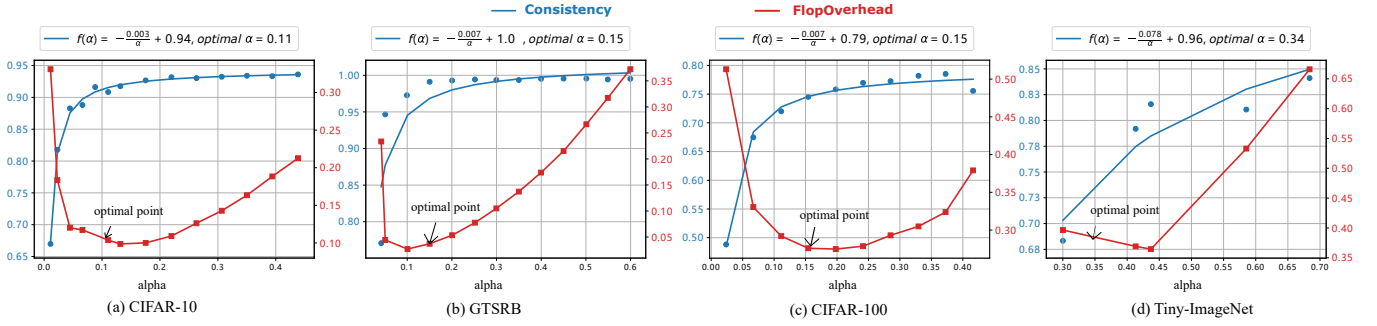[3]More fault injections are performed, and the results are similar.

Figure 6: Consistency v.s. overhead in the architecture exploration stage.

Table 4: Task simplification further shrink the overhead.

Dataset	Impact Matrix	Start Consistency	Before TaskSim.		After TaskSim		k
			O(C)	Wcov	O(C)	Wcov	
CIFAR-10	non-uniform	91.62%	9.18%	86.94%	6.88%	86.12%	8
	uniform		9.18%	75.90%	9.11%	75.90%	9
GTSRB	non-uniform	98.75%	2.68%	98.46%	1.94%	98.23%	23
	uniform		2.68%	98.15%	2.56%	98.15%	33
CIFAR-100	non-uniform	75.82%	27.48%	67.29%	24.02%	66.92%	83
	uniform		27.48%	74.33%	27.42%	74.33%	99
Tiny-ImageNet	non-uniform	79.19%	36.91%	76.40%	35.56%	75.19%	186
	uniform		36.91%	78.03%	36.84%	78.02%	198

improves with the increasing size of checker DNNs. Also, as we can observe, there is a turning point on the computational overhead curve. Before the turning point, the re-computation dominates the computational overhead O(C), and after which, the checker models' computational cost dominates the O(C).

First, compared to the architecture with the highest consistency (i.e., duplication), the optimized architecture can greatly reduce the overhead. For CIFAR-10, GTSRB, CIFAR-100, and Tiny-ImageNet, 91.82%, 97.32%, 72.52% and 63.09% overhead can be saved with 8.38%, 1.25%, 24.18% and 20.81% consistency degradation, respectively (see Table 4). We also observe that the consistency values vary for different datasets. After manual checking, we found the consistency values of the optimized architecture on the training set are almost 100%, but it generalizes differently during inference for the four data sets.

Second, the relation between consistencies, architecture sizes, and the optimal point is well captured by THEOREM 4.1 . For example, the resulting consistency function for CIFAR-10 is $f(\alpha) = -\frac{0.003}{\alpha} + 0.94$, and hence the optimal point is when $\alpha = 0.11$ (recall that the optimal point is $\sqrt[3]{\frac{a}{x}}$). This is compatible with the red curve where $\alpha = 0.11$ almost renders the minimal computational overhead. Similarly, the consistency function for GTSRB is $f(\alpha) = -\frac{0.007}{\alpha} + 1.0$ and the calculated optimal point is $\alpha = 0.15$, which is also compatible with the red curve. Therefore, the initial checker DNN architecture can be efficiently found by the proposed method.

6.3 Effectiveness of Task Exploration

In this part, we first evaluate the weighted coverage $Wcov.$ and overhead $O(C)$ under various simplified classes k. Then, we study how the risk impact matrix affects task simplification.

First, we observe that task simplification can significantly reduce the overhead of DeepDyve with little coverage degradation. In Figure 7, we use the checker DNN design obtained from Section 6.2 and trace the weighted coverage $Wcov.$ and overhead $O(C)$ during the agglomerative class clustering process. Table 4 shows the final task simplification results. Through task simplification, we can save the overhead by $(9.18\% - 6.88\%)/9.18\% = 25.05\%$, 27.61%, 12.60%, 3.38% for CIFAR-10, GTSRB, CIFAR-100, and Tiny-ImageNet, at the cost of 0.9%, 0.2%, 0.5%, 1.6% coverage degradation, respectively.

We can also observe that fault coverage and overhead vary a lot under different impact matrix configurations. First, the initial weighted coverage values before task simplification is different for uniform and non-uniform settings, because the I_{ij} term in the definition of weighed coverage given by Equation 4 varies. Second, we observe much more overhead savings can be achieved in the non-uniform case. For example, the overhead saving can be improved from 0.76% to 25.05% on CIFAR-10 when changing to non-uniform impact matrix. This is because, the impact of classes with low precision is set as lower values under such circumstances, which provides more opportunities for task simplification. In other words, a reasonable impact matrix is beneficial for protection with DeepDyve and hence is highly recommended.

6.4 DeepDyve vs. Threshold Checking

In this section, we compare the performance of DeepDyve with the Threshold Checking scheme proposed in [23]. We experiment under both random fault attack and BitFlip Attack (BFA) settings. We show the results in Table 5, including the false-positive rate (FPR), false-negative rate (FNR), computational overhead (O(C)), and the weighted coverage (Wcov.).

First, we observe DeepDyve significantly outperforms Threshold Checking in terms of Wcov., which in turn leads to smaller FNR. As discussed in Section 2, most intermediate activation values locate in the normal range even under fault attack, especially for the quantized DNN case. Hence, most faults are missed with Threshold Checking, but they can be detected by DeepDyve.

Second, in most cases, we can observe both DeepDyve and Threshold Checking performs better under BFA compared to random fault attacks. We manually check the internal values of DNN under fault attack and find that the magnitude of value change is larger in BFA than in random fault attacks, thereby making fault detection easier.
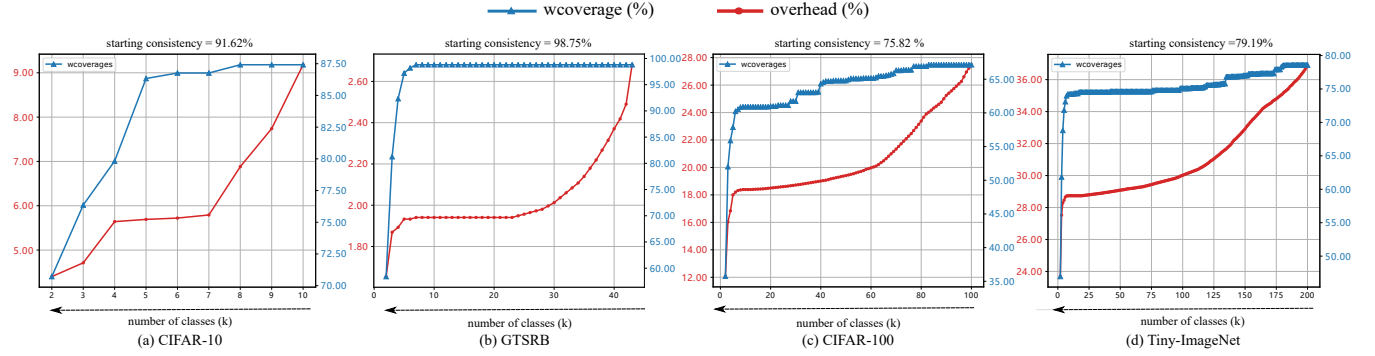
Figure 7: Tracing of the fault coverage and overhead change in task simplification process.

Table 5: Comparison between DeepDyve and threshold checking.

Dataset	Impact Matrix	Random Fault Attack								BFA							
		Threshold Checking				DeepDyve				Threshold Checking				DeepDyve			
		FPR	FNR	O(C)	Wcov.	FPR	FNR	O(C)	Wcov.	FPR	FNR	O(C)	Wcov.	FPR	FNR	O(C)	Wcov.
CIFAR-10	non-uniform	0.04%	96.24%	-	**4.10%**	0.00%	43.29%	6.88%	**82.98%**	0.04%	33.45%	-	**66.39%**	0.00%	1.01%	6.88%	**98.93%**
	uniform	0.04%	90.87%	-	**9.12%**	0.00%	24.06%	9.11%	**75.94%**	0.04%	55.52%	-	**44.48%**	0.00%	2.15%	9.11%	**97.85%**
GTSRB	non-uniform	0.00%	99.72%	-	**0.3%**	0.00%	16.53%	1.94%	**95.49%**	0.00%	100.00%	-	**0.00%**	0.00%	0.90%	1.94%	**99.75%**
	uniform	0.00%	96.85%	-	**3.15%**	0.00%	5.07%	2.56%	**94.93%**	0.00%	99.88%	-	**0.12%**	0.00%	0.79%	2.56%	**99.21%**
CIFAR-100	non-uniform	0.02%	65.11%	-	**33.48%**	0.00%	31.51%	24.02%	**74.67%**	0.02%	14.50%	-	**86.61%**	0.00%	1.84%	24.02%	**99.50%**
	uniform	0.02%	86.99%	-	**13.01%**	0.00%	19.21%	27.42%	**80.79%**	0.02%	1.05%	-	**98.95%**	0.00%	0.37%	27.42%	**99.63%**
Tiny-ImageNet	non-uniform	0.02%	69.31%	-	**30.56%**	0.00%	19.46%	35.19%	**82.00%**	0.02%	0.00%	-	**100.00%**	0.00%	0.05%	35.19%	**99.94%**
	uniform	0.06%	87.34%	-	**12.65%**	0.00%	17.02%	36.84%	**82.98%**	0.06%	0.08%	-	**99.92%**	0.00%	0.02%	36.84%	**99.98%**

Third, the FPR of Threshold Checking is above zero while that of our DeepDyve system is zero. For example, the FPR of Threshold Checking for CIFAR-10 dataset is 0.04%. As discussed in Section 2, threshold Detection sets the thresholds as 1.1 times the maximum and a minimum of each layer's normal activation values on the training set. On the testing set, there are few exceptions where the activation values are beyond this range. In contrast, in the normal execution of DeepDyve, the comparator's false positives, which are caused by inconsistencies between task and checker models, are subject to re-computation, and hence, the system's false positives are guaranteed to be zero.

6.5 Impact of Model Accuracy

Previous experiments suggest that the overhead of DeepDyve for CIFAR-100 and Tiny-ImageNet dataset are quite high, even after task simplification. This is because the per-class accuracy of the task model on these two datasets varies and some of them are very low, as shown in Figure 8. In safety-critical applications, a class is deserved to be protected only when its accuracy is high enough. Considering the above, we conduct a case study on CIFAR-100 and let the impact of 75% of the classes with the lowest precision to be zero.

Previously, the computational overhead induced by the checker DNN before task simplification was 27.48% (See Table 4). With the above setting, as we only care classes with non-zero impact, we let the comparator only check the inconsistencies of these classes. Given this, the overhead induced by the checker DNN before task simplification is 16.17%. Task simplification can further reduce the overhead from 16.17% to 9.88% without loss of weighted coverage (38.89% overhead savings). Also, the simplified model can reach
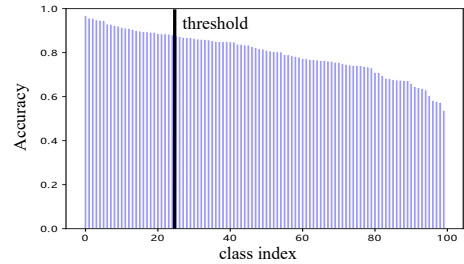

Figure 8: Per-class accuracy for CIFAR-100.

90.66% weighted coverage under random fault attack and 99.78% weighted coverage under BFA attack.

7 DISCUSSIONS

In this section, we discuss the robustness of the proposed DeepDyve architecture and its limitations.

7.1 Robustness of DeepDyve

What if the checker DNN has faults? If the checker DNN is faulty while the task DNN is correct, the final inference accuracy would remain the same, because the system would accept the output from the task model anyway. There could be extra latency. To mitigate this issue, we could leverage various hardening techniques (e.g., secure enclaves) to protect it at a reasonable cost since the checker DNN is much smaller than the task DNN model.

Attack on DeepDyve. Attackers need to create consistent faulty outputs to bypass the comparison logic of DeepDyve to successful launch their attacks.

One way to achieve this is to inject faults into the task and checker models simultaneously. However, the cost of launching such an attack in practice is very high, if not impossible. On the one hand, simultaneously injecting faults at two specific positions is difficult. For example, row-hammer attack relies on the weakness of physical memory row, and it cannot be fully controlled. On the other hand, if low-precision fault injection technique is used to inject random faults into the two DNNs, the probability distribution of the faulty output of a DNN is given by its risk probability matrix, and the probability of two DNNs' outputs happens to be the same is given by

$$P_{collision} = \sum_i^N p_i * q_i, \tag{17}$$

where p_i and q_i are the probabilities of task DNN and checker DNN generating the same output i, respectively, and N is the number of classes. Obviously, this value decreases with the increase of classes. The $P_{collision}$ values are 8.9%, 2.77%, 0.87%, and 0.49% for CIFAR-10, GTSRB, CIFAR-100, and Tiny-ImageNet, respectively. Given that the probability of DNNs generating wrong outputs under random faults P_{error} is extremely low. The possibility for such attack to succeed is negligible.

Another way to successfully launch fault injection attacks is to target those inconsistent cases and make them consistent with faulty result. To achieve this objective, however, attackers need to be able to tell whether an incoming data is consistent at runtime and perform fault injection before its inference is finished. This is a daunting objective to achieve, especially considering the usual long preparation time for fault injection.

7.2 Limitations and Future Work

DeepDyve brings latency and reduces throughput due to recomputation, which needs to be considered when performing real-time tasks. For example, there is about 1% of throughput loss for the above DeepDyve model constructed on the GTSRB dataset. This overhead can be reduced when the consistency rate between the checker DNN and the task DNN models increases.

Also, our current evaluation is only on the 8-bit integer (INT-8) data representation. One reason is that INT-8 is a popular data type and supported by well known deep learning frameworks and tool-chains, such as Pytorh [4], Tensorflow[5], NVIDIA®TensorRD [6], and Xilinx®DNNDK [7]. Popular hardware platforms like TPU [15] support integer operations as well. The second reason is that the Pytorch 1.3 (the one we use) and the code from BFA attack [8] only supports INT-8 quantization in its current version. We shall extend our work to study the impact of different data types in the future.

---

[4]https://pytorch.org/docs/stable/quantization.html
[5]https://www.tensorflow.org/lite/performance/post_training_quantization
[6]https://developer.nvidia.com/tensorrt
[7]https://www.xilinx.com/products/design-tools/ai-inference/edge-ai-platform.html#dnndk
[8]https://github.com/elliothe/BFA

8 RELATED WORKS

DNNs have become an enabling technology for many safety-critical applications, and hence there is a growing research interest for fault-tolerant DNN designs.

In the literature, some works focused on improving fault-tolerance of DNN systems built with unreliable *resistive random-access memory* (RRAM) [3, 25, 41], an emerging device with huge energy efficiency benefits. However, this technology is still in its infancy and is not applicable for safety-critical AI applications. Existing fault-tolerant solutions for DNN systems built with conventional CMOS devices can be categorized as *passive* solutions or *active* solutions.

Passive fault-tolerant solutions reduce the probability of failure occurrences for a certain level at the design stage, including *fault-mitigation* techniques for the DNN design itself and *fault-masking* solutions with redundant circuitries. To be specific, fault-mitigation techniques include: (i) *Resilient training* [7, 16, 43, 46], which explicitly considers hardware faults during the DNN training phase; (ii) *Resilient architecture design* [5, 6, 8, 37], which tries to find an error-resilient network architecture or inserts redundancies for those critical elements in the system; (iii) *Device hardening* [2, 23], which tries to protect some critical storage elements in the system by selectively hardening them. The above solutions require significant design effort, and they usually cannot provide sufficient fault coverage. Fault-masking techniques are conceptually simple. We could use *error-correcting codes* (ECC) to protect memory elements [35] and *triple-modular redundancy* (TMR) to protect computational units [29, 42]. However, the corresponding hardware overhead is exceptionally high.

Active fault-tolerant solutions detect and recover from faults by re-organizing the system in real-time. To achieve active fault-tolerance, we need to be able to detect faults online. There are a few solutions in this direction [23, 24, 38]. Among these techniques, [23, 38] has small hardware overhead, but their fault detection capabilities are limited, especially for quantized DNNs; the fault coverage of the replication-based solution in [24] is comparable to DeepDyve, but its overhead is much higher.

9 CONCLUSION

In this paper, we proposed DeepDyve, a novel dynamic verification technique against fault injection attacks on DNN systems. By introducing a far simpler and smaller checker DNN into the system, DeepDyve significantly outperforms state-of-the-art solutions, reducing 90% risks at around 10% computational overhead.

REFERENCES

[1] Todd M Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*. IEEE, 196–207.
[2] Arash Azizimazreah, Yongbin Gu, Xiang Gu, and Lizhong Chen. 2018. Tolerating soft errors in deep learning accelerators with reliable on-chip memory designs.

In *IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 1–10.

[3] Lerong Chen, Jiawen Li, Yiran Chen, Qiuping Deng, Jiyuan Shen, Xiaoyao Liang, and Li Jiang. 2017. Accelerator-friendly neural-network training: Learning variations and defects in RRAM crossbar. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. European Design and Automation Association, 19–24.

[4] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *IEEE Journal of Solid-State Circuits (JSSC)*. IEEE, 127–138.

[5] C-T Chiu, Kishan Mehrotra, Chilukuri K Mohan, and Sanjay Ranka. 1993. Robustness of feedforward neural networks. In *IEEE International Conference on Neural Networks (ICNN)*. IEEE, 783–788.

[6] L-C Chu and Benjamin W Wah. 1990. Fault tolerant neural networks with hybrid redundancy. In *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 639–649.

[7] Jiacnao Deng, Yuntan Rang, Zidong Du, Ymg Wang, Huawei Li, Olivier Temam, Paolo Ienne, David Novo, Xiaowei Li, Yunji Chen, et al. 2015. Retraining-based timing error mitigation for hardware neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 593–596.

[8] Fernando Morgado Dias, Rui Borralho, Pedro Fontes, and Ana Antunes. 2010. FTSET-a software tool for fault tolerance evaluation and improvement. In *Neural Computing and Applications (Neural. Comput. Appl.)*, Vol. 19. Springer, 701–712.

[9] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *3rd International Conference on Learning Representations (ICLR)*. OpenReview.net.

[10] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations (ICLR)*. OpenReview.net.

[11] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. 2014. Distilling the Knowledge in a Neural Network. In *NIPS Deep Learning and Representation Learning Workshop (NIPS Workshop)*. Curran Associates Inc.

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

[13] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[14] ISO. 2016. ISO-26262: Road vehicles – Functional safety. ISO, Geneva, Switzerland.

[15] Norman P Jouppi and et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–12.

[16] Sung Kim, Patrick Howe, Thierry Moreau, Armin Alaghi, Luis Ceze, and Visvesh Sathe. 2018. MATIC: Learning around errors for efficient low-voltage neural network accelerators. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[17] Yoongu Kim, Daly Ross, and et. al. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 361–372.

[18] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. In *arXiv preprint arXiv:1806.08342*.

[19] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.

[20] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial Machine Learning at Scale. In *5th International Conference on Learning Representations (ICLR)*. OpenReview.net.

[21] Ya Le and Xuan Yang. 2015. Tiny imagenet visual recognition challenge. *Stanford CS 231N Course*.

[22] Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems (NIPS)*. Curran Associates Inc., 598–605.

[23] Guanpeng Li and et al. 2017. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 8:1–8:12.

[24] Yu Li, Yannan Liu, Min Li, Ye Tian, Bo Luo, and Qiang Xu. 2019. D2NN: a fine-grained dual modular redundancy framework for deep neural networks. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*. ACM, 138–147.

[25] Chenchen Liu, Miao Hu, John Paul Strachan, and Hai Li. 2017. Rescuing memristor-based neuromorphic design with high defects. In *54th*

[26] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. 2017. Fault injection attack on deep neural network. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 131–138.

[27] Bo Luo, Yannan Liu, Lingxiao Wei, and Qiang Xu. 2018. Towards imperceptible and robust adversarial example attacks against neural networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 1652–1659.

[28] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *6th International Conference on Learning Representations (ICLR)*. OpenReview.net.

[29] Hamid Reza Mahdiani, Sied Mehdi Fakhraie, and Caro Lucas. 2012. Relaxed fault-tolerant hardware implementation of neural networks in the presence of multiple transient errors. *IEEE transactions on neural networks and learning systems (TNNLS)* 23, 1215–1228.

[30] Masato Matsubayashi, Akashi Satoh, and Jun Ishii. 2016. Clock glitch generator on SAKURA-G for fault injection attack against a cryptographic circuit. In *IEEE 5th Global Conference on Consumer Electronics (GCCE)*. IEEE, 1–4.

[31] Dongyu Meng and Hao Chen. 2017. Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the ACM SIGSAC conference on computer and communications security (CCS)*. ACM, 135–147.

[32] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *Proceedings of the ACM on Asia conference on computer and communications security (Asia CCS)*. ACM, 506–519.

[33] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. 2019. Bit-Flip Attack: Crushing Neural Network with Progressive Bit Search. In *IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, 1211–1220.

[34] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. 2018. Ares: A framework for quantifying the resilience of deep neural networks. In *55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[35] Brandon Reagen, Paul Whatmough, and et. al. Adolf. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 267–278.

[36] Yolan Romailler and Sylvain Pelissier. 2017. Practical fault attack against the Ed25519 and EdDSA signature schemes. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 17–24.

[37] Christoph Schorn, Thomas Elsken, Sebastian Vogel, Armin Runge, Andre Guntoro, and Gerd Ascheid. 2020. Automated design of error-resilient and hardware-efficient deep neural networks. In *Neural Computing and Applications (Neural. Comput. Appl.)*. Springer, 1–19.

[38] Christoph Schorn, Andre Guntoro, and Gerd Ascheid. 2018. Efficient On-Line Error Detection and Mitigation for Deep Neural Network Accelerators. In *International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*. Springer, 205–219.

[39] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. 2012. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks* 32, 323–332.

[40] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, Vol. 97. PMLR, 6105–6114.

[41] Lixue Xia, Mengyun Liu, Xuefei Ning, Krishnendu Chakrabarty, and Yu Wang. 2017. Fault-tolerant training with on-line fault detection for RRAM-based neural computing systems. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC)*. ACM, 33:1–33:6.

[42] Zheyu Yan, Yiyu Shi, Wang Liao, Masanori Hashimoto, Xichuan Zhou, and Cheng Zhuo. 2020. When Single Event Upset Meets Deep Neural Networks: Observations, Explorations, and Remedies. In *25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 163–168.

[43] Lita Yang and Boris Murmann. 2017. SRAM voltage scaling for energy-efficient convolutional neural networks. In *18th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 7–12.

[44] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. 2020. DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips. In *29th USENIX Security Symposium (USENIX Security)*. USENIX Association, 1463–1480.

[45] Pu Zhao, Siyue Wang, Cheng Gongye, Yanzhi Wang, Yunsi Fei, and Xue Lin. 2019. Fault Sneaking Attack: a Stealthy Framework for Misleading Deep Neural Networks. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC)*. ACM, 165.

[46] He Zhezhi, Rakin Adnan, Siraj, Li Jingtao, Chakrabarti Chaitali, and Deliang Fan. 2020. Defending and Harnessing the Bit-Flip based Adversarial Weight Attack. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 14083–14091.