BinFI: An Efficient Fault Injector for Safety-Critical Machine Learning Systems

Zitao Chen
zitaoc@ece.ubc.ca
University of British Columbia

Guanpeng Li
gpli@ece.ubc.ca
University of British Columbia

Karthik Pattabiraman
karthikp@ece.ubc.ca
University of British Columbia

Nathan DeBardeleben
Los Alamos National Laboratory
ndebard@lanl.gov

ABSTRACT

As machine learning (ML) becomes pervasive in high performance computing, ML has found its way into safety-critical domains (e.g., autonomous vehicles). Thus the reliability of ML has grown in importance. Specifically, failures of ML systems can have catastrophic consequences, and can occur due to soft errors, which are increasing in frequency due to system scaling. Therefore, we need to evaluate ML systems in the presence of soft errors.

In this work, we propose *BinFI*, an efficient fault injector (FI) for finding the safety-critical bits in ML applications. We find the widely-used ML computations are often *monotonic*. Thus we can *approximate* the error propagation behavior of a ML application as a monotonic function. *BinFI* uses a binary-search like FI technique to pinpoint the safety-critical bits (also measure the overall resilience). *BinFI* identifies 99.56% of safety-critical bits (with 99.63% precision) in the systems, which significantly outperforms random FI, with much lower costs.

KEYWORDS

Error Resilience, Machine Learning, Fault Injection

ACM Reference Format:
Zitao Chen, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. 2019. *BinFI*: An Efficient Fault Injector for Safety-Critical Machine Learning Systems . In *Proceeding of SC19, November 17-22, 2019, Denver, CO*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The past decade has seen the massive adoption of machine learning (ML) in a diverse set of areas [36, 47, 71, 72]. ML has also found its way into the high performance computing (HPC) domain, where the high computing capacity empowers ML to manage the large volume of scientific and engineering data [35, 56, 65, 81]. Many of these ML applications are safety-critical [19, 26, 60, 80, 81]. One emerging example is autonomous vehicles (AVs), in which the *high throughput, low latency, high reliability* requirements make the hardware of these applications rival those of some supercomputers. For example, Nvidia Drive AGX Xavier, a system-on-chip (SoC) designed for AVs application, is able to deliver 30 TOPS (trillion operations per second) of performance at 30W [10], much like HPC systems [49]. While we focus on AVs in this paper, similar trends apply for ML applications in domains such as health-care [80, 81].

As the trends toward exascale computing and AVs continue to grow, the ability of ML to deliver high performance critically depends on the reliability of the system [23, 70]. The ISO 26262 standard for functional safety of road vehicles specifies that there can be no more than 10 FIT (Failures in Time), which is 10 failures in a billion hours of operation [8, 49]. A recent study of failures in AVs shows that faults related to ML systems are the major culprits, often requiring the fallback of human driver intervention [18]. Many of these failures are due to hardware transient faults, also known as *soft errors*, which are typically caused by high-energy particles due to cosmic rays that interact with electronic components (e.g., a terrestrial neutron strike may cause current spikes in logic circuits or storage elements, and subsequently lead to a bit flip). Moreover, soft errors are increasing in frequency as system scale increases (especially in HPC applications [23, 70, 73]) and they could lead to undesirable consequences in ML systems [25, 49, 62]. In addition, hardware faults can also be *deliberately* induced by malicious attackers (e.g., selectively flipping specific bits), which can also cause significant performance degradation in DNNs [38].

Traditional techniques (e.g., replicating hardware components [53]; instruction duplication [51]) incur high overheads in hardware cost, energy and performance, which make them impractical to be deployed in HPC ML systems [49]. For example, an AV ready for real-world deployment should be able to process the driving data within a few milliseconds (per iteration) [2, 15]. Deploying traditional protection techniques may lead to delays in response time, which may in turn lead to reaction-time-based accidents [18]. Therefore, we need to identify the parts of the ML systems that are the most sensitive to *hardware transient faults*, and selectively protect the sensitive parts at low costs.

In this paper, we focus on the problem of identifying the safety-critical bits in ML applications. These are the bits in the ML program, which if affected by hardware transient faults, result in a safety-condition violation. A well-established approach to experimental resilience assessment is random fault injection (FI), which works by randomly sampling from a set of fault locations, and injecting the faults into the program to obtain a statistically significant estimate of their overall resilience. Random FI has been used on ML applications for overall resilience assessment [49, 62], but it is unfortunately not suitable for identifying the safety-critical bits in the

SC19, November 17-22, 2019, Denver, CO, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

program due to two reasons. First, because random FI relies on statistical sampling, it is not guaranteed to cover all the safety-critical bits. Second, the safety-critical bits are often clustered in the state space, and random sampling is unable to find them (e.g., Fig. 3).

The only known approach to identify the safety-critical bits of a ML program is exhaustive FI, which involves flipping each bit of the program, and checking if it resulted in a safety violation. Unfortunately, exhaustive FI incurs huge performance overheads, as only one fault is typically injected in each trial (for controllability). The time taken by exhaustive FI is therefore directly proportional to the number of bits in the program, which can be very large.

In this paper, we propose an efficient FI approach, which can identify the safety-critical bits in ML applications, and also measure the overall resilience, with reasonable overheads. The key insight of our approach is that the functions used in ML applications are often tailored for specific purposes. For example, for a network to classify an image of a vehicle, the ML computations would be designed in a way that they can produce larger response upon the detection of the vehicular feature in the image, while keeping the response to irrelevant features small. This results in the functions exhibiting *monotonicity* based on how compatible is the input with the target (e.g., vehicular features), and the *composition* of these ML functions can be *approximated* as a single *monotonic* composite function (Section 3.2). The monotonicity of the function helps us prune the FI space and efficiently identify the safety-critical bits. *Analogous to how binary search on a sorted array has an exponential time reduction compared to linear search, our approach results in an exponential reduction in the FI space of ML applications to identify the safety-critical bits, compared to exhaustive FI.* Therefore, we call our approach *Binary fault injection* or BinFI (in short).

Prior work has attempted to find safety violations in ML programs for design/software bugs through random testing approaches [31, 54, 59, 78]. However, these papers do not examine the effects of hardware faults, more specifically, soft errors. Other papers have attempted to prune the FI space for general-purpose programs [33, 69], or even eliminate FI altogether [51]. Unfortunately, these papers do not consider the specific properties of ML programs. *To the best of our knowledge,* BinFI *is the first fault injection technique to efficiently find safety violations in generic ML applications due to soft errors.*

Our main contributions in this paper are as follows:

- Analyze the common operations used in ML applications and identify that many of them exhibit *monotonicity*. We approximate the composition of these functions as a monotonic function.
- Present *BinFI*, a fault injection approach that leverages the approximated monotonic composite function, to find the safety-critical bits, and also measure the overall resilience of the program.
- Extend an open-source FI tool, *TensorFI* to incorporate *BinFI*, and evaluate it over 8 ML applications with a total of 6 datasets (including a real-world driving dataset).

Our evaluation shows that *BinFI* can identify 99.56% of the critical bits with 99.63% precision, which significantly outperforms random FI. Further, it can also accurately estimate the overall SDC probability of the application. Finally, *BinFI* incurs only around 20% of the performance overhead of exhaustive FI, which is the only other way to identify most of the critical bits in a program (to our knowledge) i.e., it obtains a speedup of 5X over exhaustive FI.

While the coverage of *BinFI* is not 100% in identifying safety-critical bits, it presents an attractive alternative to traditional exhaustive fault injection due to its significantly low overheads. Further, the proportion of critical bits missed by *BinFI* is less than 0.5%, which is acceptable in many situations given that soft errors are relatively rare events, and that not all bit-flips would lead to safety violations in practice [43]. Therefore, *BinFI* is the first step towards exploring the space of techniques that trade-off resilience for performance overheads in safety-critical ML applications.

2 BACKGROUND AND FAULT MODEL

We start by providing an overview of the fault injection tool we used. We then discuss the requirements for AVs, and finally present the fault model we assume.

2.1 TensorFlow and Fault Injection Tool

In our work, we use an open-source fault injector called TensorFI, which is a configurable FI tool to inject faults in TensorFlow applications [50]. We choose it as it supports FI experiments on ML algorithms implemented using the TensorFlow framework [16], which is the most popular ML framework in use today [14][1]. The main advantage of TensorFlow is that it abstracts the operations in a ML program as a set of operators and a dataflow graph - this allows programmers to focus on the high-level programming logic.

There are two main components in the TensorFlow dataflow graph: (1) *operator* is the computational unit (e.g., matrix multiplication) and *tensor* is the data unit. Users can build the ML model using the built-in operators or define their customized operators. TensorFI duplicates the TensorFlow graph with *customized* operators [50], which are designed to not only be able to perform computation as standard operators do, but also inject faults at runtime. TensorFI is provided as a library so that users can easily integrate TensorFI into their TensorFlow programs. It also allows different configuration (e.g., how and where to inject fault) options through a YAML interface.

2.2 AVs Requirements

Autonomous vehicles (AVs) are complex systems that use ML to integrate data from various electronic components (e.g., LiDAR) and deliver real-time driving decisions. AVs entail several requirements: (1) *high throughput* (e.g., large amounts of data must be processed as they arrive from the sensors [1, 15]), and (2) *low latency* (e.g., apply the brakes upon the detection of a pedestrian in front of the vehicle within a few *ms*) [2, 15]. These requirements present significant challenges for reliability in AV applications.

As mentioned, AVs reliability is mandated by stringent regulation - no more than 10 FIT as governed by ISO 26262 safety standard. There are two kinds of faults that can cause this standard to be violated [2], namely (1) systematic faults, and (2) transient faults. The former are caused by design defects in hardware and software, while the latter by cosmic rays and electromagnetic disturbances. Systematic faults in AVs can be mitigated at design time, and have been well explored in the literature [59, 67, 78]. Hardware transient faults, on the other hand, need runtime mitigation and are less

[1] Our approach is also applicable for applications using other ML frameworks such as PyTorch, Keras, etc., as they are similar to TensorFlow in structure.

studied in the context of AVs. FIT due to soft errors is orders of magnitude higher than the 10 FIT requirement [49]. Therefore, it is important to study soft errors' effect on AV's reliability.

2.3 Fault Model

In this paper, we consider hardware transient faults (i.e., soft errors) that randomly occur during the execution of ML programs. The errors are caused by different sources such as alpha particles and electro-magnetic effects. In our fault model, we assume faults occur in the processor's data path, i.e., in pipeline registers and ALUs. We assume that main memory, cache and the register file are protected by ECC or parity, and hence we do not consider faults that originate in them. This is a common assumption made in fault injection studies [17, 29, 51]. We also assume that faults do not occur in the processor's control logic, as that constitutes only a small portion of the total area of the processor. Note that our fault model only considers faults that are not masked before reaching the software layer, as masked faults do not affect the application's execution.

Since TensorFI operates on TensorFlow graphs, we map the fault occurrence to the interface of the operators in the graph. This is because the details of each operation are hidden from the ML program, and are also platform specific (e.g., the GPU version of these operators are often different from the CPU version). As a result, we inject faults directly to the output value of operators in ML programs - this method is in line with prior work [20, 50, 51].

We follow the one-fault-per-execution model as soft errors are relatively rare events with respect to the typical time of a program's execution - again this is a common assumption in the literature [17, 27, 51, 79]. Soft errors can manifest in the software layer as single or multiple-bit flips. However, recent work [20, 68] has shown that multiple bit flip errors result in similar error propagation patterns and SDC probabilities as single-bit flip errors (at the program level), showing that studying single-bit flip fault injection is sufficient for drawing conclusions about error resilience.

We assume that faults do not modify the state/structure of the model (e.g., change the model's parameters [42, 54]), nor do we consider faults in the inputs provided to the model (e.g., sensor faults such as brightness change of the image [67, 78]), as they are outside the scope of the technique. Finally, we only consider faults during the inference phase of ML programs as ML models are usually trained once offline, and the inference phase is performed repeatedly in deployment (typically, hundreds of thousands of times in AVs), which makes them much more prone to soft errors.

3 METHODOLOGY

We consider Silent Data Corruption (SDC) as a mismatch between the output of a faulty program and that of a fault-free program execution. For example, in classifier models, an image misclassification *due to soft errors* would be an SDC. We leave the determination of whether an SDC is a safety violation to the application, as it depends on the application's context (see Section 4). Unlike traditional programs, where faults can lead to different control flows [33, 51, 58], faults in ML programs only result in numerical changes of the data within the ML models (though faults might also change the execution time of the model, this rarely happens in practice as the control flow is not modified).

Critical bits are those bits in the program where the occurrence of fault would lead to an SDC (e.g., unsafe scenarios in safety-critical applications). Our goal is to efficiently identify these critical bits in ML programs without resorting to exhaustive fault injection into every bit.

We first provide an example of how faults propagate in ML program in Section 3.1. We then define the terms *monotonicity* and *approximate monotonicity* that we use throughout the paper. Then we present our findings regarding the monotonicity of the functions used in common ML algorithms in Section 3.3, so that we can model the *composition* of *all* the monotonic functions involved in fault propagation *either* as a monotonic *or* approximately monotonic function (Section 3.4). Finally, we show how we leverage the (approximate) monotonicity property to design a binary-search like FI algorithm to efficiently identify the critical bits, in Section 3.5.

3.1 Error Propagation Example

The principle of error propagation in different ML models is similar in that a transient fault corrupts the data, which becomes an erroneous data, and will be processed by all the subsequent computations until the output layer of the model. In this section, we consider an example of error propagation in the k-nearest neighbor algorithm (kNN), in Fig. 1 (k=1). The program is written using TensorFlow, and each TensorFlow operator has a prefix of *tf* (e.g., tf.add). We use this code as a running example in this section.

We assume that the input to the algorithm is an image (*testImg*), and the output is a label for the image. Line 1 calculates the negative value of the raw pixel in *testImg*. The program also has a set of images called *neighbors*, whose labels are already known; and the goal of the program is to assign the label from one of the *neighbors* (called nearest neighbor) to the *testImg*. Line 2 computes the relative distance of *testImg* to each of *neighbors*. Line 3 generates the absolute distances and line 4 summarizes the per-pixel distance into a total distance. Line 5 looks for the index of the nearest neighbor, whose label is the predicted label for *testImg*.

```
1 negPixel = tf.negative(testImg)
2 relativeDistance = tf.add(neighbors, negPixel)
3 absDistance = tf.abs(relativeDistance)
4 distance = tf.reduce_sum(absDistance, reduction_indices=1)
5 nearestNeighbor = tf.arg_min(distance, 0)
```

Figure 1: An example of error propagation in kNN model (k=1), fault occurs at the tf.add operator - line 2.

Assume a fault occurs at the add operator (line 2) and modifies its output *relativeDistance*. If each image's dimension is $(28, 28)$, then the result from the tf.add operator is a matrix with shape $(|N|, 784)$, where $|N|$ is the number of neighbors and each vector with 784 elements corresponds to each neighbor. If the i_{th} image is the nearest neighbor, we have $dis_i < dis_j, \forall j \in |N|, i \neq j$, where dis_i is the distance of the i_{th} image to the test image. The fault might or might not lead to an SDC - we consider both cases below.

SDC: The fault occurs at $(i, y), y \in [1, 784]$ (i.e., corresponding to the nearest neighbor), which incurs a positive value deviation (e.g., flipping 0 to 1 in a positive value). The fault would increase dis_i, which indicates a potential SDC as dis_i might no longer be the smallest one among $dis_j, j \in |N|$. Similarly, the fault at $(j, y), y \in [1, 784]$ incurs a negative deviation and may result in an SDC.

No SDC: The fault at $(i, y), y \in [1, 784]$ incurs a negative value deviation, thus decreasing dis_i. This will not cause SDC since dis_i is always the nearest neighbor. Similarly, a fault incurring positive deviation at $(j, y), y \in [1, 784]$ would always be masked.

3.2 Monotonicity and Approximate Monotonicity

We now define the terms monotonicity and approximate monotonicity that we use in this paper.

- **Non-strictly monotonic function:** A non-strictly monotonic function is either *monotonically increasing*: $f(x_i) \geq f(x_j), \forall x_i > x_j$; or *monotonically decreasing*: $f(x_i) \leq f(x_j), \forall x_i > x_j$. We say a function has *monotonicity* when it is non-strictly monotonic.
- **Approximately monotonic:** We call a function *approximately monotonic* if it is non-strictly monotonic in a non-trivial interval, i.e., the function does not exhibit both monotonically increasing and decreasing interchangeably (e.g., Sine function). We say a function has *approximate monotonicity* when it is approximately monotonic. For example, $f(x) = 100 * max(x - 1, 0) - max(x, 0)$ is monotonically increasing when $x > 1$, but not when $x \in (0, 1)$. Hence, we consider $f(x)$ is *approximately monotonic*.

Error propagation (EP) function: We define EP function, which is a *composite function* of those functions involved in propagating the fault from the fault site to the model's output. For example, there are MatMul (matrix multiplication) and ReLu (rectified linear unit [57]) following the occurrence of the fault, in this case EP function is the composite function of both functions: $EP(x) = ReLu(MatMul(x_{org} - x_{err}, w))$, where w is the weight for the MatMul computation, x_{org} and x_{err} are the values before and after the presence of fault. Thus $x_{org} - x_{err}$ is the deviation caused by the fault at the fault site. Note that we are more interested in the *deviation* caused by the fault rather than the value of the affected data (x_{err}). Thus, the *input* to EP is the bit-flip deviation at the fault site and the *output* of EP is the outcome deviation by the fault at the model's output.

The main observation we make in this paper is that most of the functions in ML model are monotonic, as a result of which the EP function is *either monotonic or approximately monotonic*. The main reason for the monotonicity of ML functions (and especially DNNs) is that they are designed to recognize specific features in the input. For example, the ML model in Fig. 2 is built for recognizing the image of digit 1, so the ML computation is designed in a way that it will have stronger responses to images with similar features as digit 1. Specifically, the ML computation would generate larger output, if the image has stronger features that are consistent with the target. In Fig. 2, the output of the three inputs increases as they exhibit higher consistency (values in the middle column for each input) with the ML target. And the final output by the model is usually determined by the numerical magnitude of the outputs (e.g., larger output means higher confidence of the image to be digit 1).

The EP function can be monotonic or approximately monotonic, based on the ML model. This (approximate) monotonicity can help us to prune the fault injection space of the technique. For simplicity, let us assume the EP function is monotonic. We can model the EP function as: $|EP(x)| \geq |EP(y)|, (x > y \gg 0) \cup (x < y \ll 0)$, x, y are faults at the bits of both 0 or 1 in the same data (Section 3.4 has more details). We can therefore expect the magnitude of the deviation caused by larger faults (in absolute value) to be greater than those by smaller faults. Further, a larger deviation at the final output is more likely to cause an SDC. Based on the above, we can first inject x. If it does not lead to an SDC, we can reason that faults from lower-order bits y *will not* result in SDCs without actually simulating them, as these faults would have smaller outcomes.

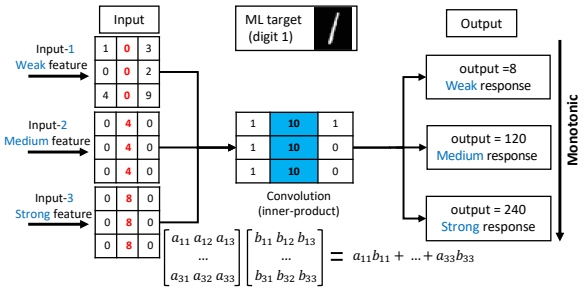

Figure 2: An example of how a ML model exhibits monotonicity for different inputs. Assume the computation is a simple one-time convolution (inner-product).

In practice, the EP function is often approximately monotonic (rather than monotonic), especially in real-world complex ML models such as DNNs. Our approach for pruning the fault injection space remains the same. This leads us to some inaccuracy in the estimation due to our approximation of monotonicity. Nonetheless, we show later in our evaluation that this inaccuracy is quite small.

3.3 Common ML Algorithms and Monotonicity

In this section, we first survey the major *ML functions* within the state-of-art ML models in different domains, and then discuss their monotonicity. Most of these models are comprised of DNNs.

Image classification: LeNet [48], AlexNet [47], VGGNet [72], Inception [75–77], ResNet [36]. Some of these networks [36, 47, 72, 76] are the winning architectures that achieve the best performance on the ILSVRC challenges [24] from 2012 to 2017.

Object detection: Faster-RCNN [66], YoLo [63], DarkNet [64]. These networks produce outstanding performance in object detection (e.g., can detect over 9000 categories [64]) and we are primarily interested in the ML thread for *classification* as these networks include both object localization and classification.

Steering models: Nvidia DAVE system [19], Comma.ai's steering model [5], Rambo [12], Epoch [7], Autumn [3]. These models are the popular steering models available in the open-source community [2] [3] and are used as the benchmarks in related studies [59, 78].

Health care and others: Detection of atrial fibrillation [80]; arrythmia detection [60]; skin cancer prediction [26]; cancer report analysis [81]; aircraft collision avoidance system [44].

These tasks are important in safety-critical domains (e.g., self-driving cars, medical diagnosis). We are interested in those computations for calculating the results (e.g., those in hidden layers). Note that errors in the input layer such as during the reading the input

[2] https://github.com/udacity/self-driving-car
[3] https://github.com/commaai/research

Table 1: Major computations in state-of-art DNNs

Basic	Conv; MatMul; Add (BiasAdd)
Activation	ReLu; ELu;
Pooling	Max-pool; Average-pool
Normalization	Batch normalization (BN); Local Response Normalization (LRN)
Data transformation	Reshape; Concatenate; Dropout
Others	SoftMax; Residual function

image are out of consideration. The computations are summarized in Table 1.

We next discuss how most of the computations used in the above tasks are monotonic.

- Basic: Convolution computation (Conv) is mainly used in the kernels in DNNs to learn the features. Conv is essentially an inner-product computation: $\vec{X} \cdot \vec{W} = \sum x_i w_i, x_i \in \vec{X}, w_i \in \vec{W}$. Assuming there are two faults at the same location and $x_1, x_2 (x_1 > x_2 > 0)$ are the deviations from the two faults. The monotonicity property is satisfied as: $|x_1 w_i| \geq |x_2 w_i|$. As mentioned that we are more interested in the effect caused by the single bit-flip fault, so we do not consider data that are not affected by fault. The multiply function is monotonic, thus Conv is monotonic (similarly for matrix multiplication - MatMul).
- Activation: Activation (Act) function is often used to introduce non-linearity into the network, which is important for the network to learn non-linear complicated mappings between the inputs and outputs [34]. A widely used activation function is Rectified linear unit (ReLu) [57], defined as: $f(x) = max(0, x)$, which is monotonic. Exponential linear unit (ELu) [55] is similar to ReLu and is monotonic.
- Pooling: Pooling is usually used for non-linear down sampling. Max-pooling and average-pooling are the two major pooling functions. Max-pooling function extracts the maximum value from a set of data for down-streaming, and it is monotonic as follows: $max(x_i, x_k) \geq max(x_j, x_k), if x_i > x_j$, where x_i, x_j could be faults at different bits, and x_k denotes all the data unaffected by the fault. Similarly, average-pooling function calculates the average value from a group of data, and it is also monotonic as follows: $avg(x_i, x_k) \geq avg(x_j, x_k), if x_i > x_j$.
- Normalization: Normalization (Norm) is used to facilitate the training of the network by dampening oscillations in the distribution of activations, which helps prevent problems like gradient vanishing [41]. Local response Norm (LRN) and batch Norm (BN) are the two Norm approaches used in the above networks.
  LRN implements a form of lateral inhabitation by creating competition for big activities among neuron outputs from different kernels [47]. However, LRN does not satisfy the monotonicity property as it normalizes the values across different neurons in a way that only needs to maintain competition (relative ordering) among the neighboring neurons. Thus a larger input might generate a smaller output as long as the normalized value maintains the same relative ordering among the neighboring neurons (we also validated this by experiments). However, LRN was found to have significant limitations [72], and hence modern ML algorithms usually use BN instead of LRN [12, 19, 36, 64, 75–77].
  BN normalizes the value in a mini-batch during training phase to improve the learning, and normalization is neither necessary nor desirable during the inference as the output is expected to be only dependent on the input deterministically [41]. And thus in inference phase, BN applies the same linear transformation to each activation function given a feature map. So we can describe BN in inference phase as: $f(x) = wx + b$, where w, b are the statistics learned during training phase. BN is thus monotonic as $f'(x) = w$.
- Data transformation: There are computations that simply transform the data, e.g., reshape the matrix, and not alter the data value. Dropout is considered as an identity mapping since it performs dropout only in training phase [74]. These transforming functions are thus monotonic as: $x_i > x_j, if\ x_i > x_j$.
- Others: SoftMax [37] is often used in the output layer to convert the logit (the raw prediction generated by the ML model), computed for each class into a probability within (0,1) and the probabilities of all classes add up to 1. SoftMax function is defined as: $f(x_i) = e^{x_i} / \sum_{j=1}^{J} e^{x_j} (for\ i = 1, \hat{a}\check{A}\c{e}, J)$, where x_i the predicted value for different class (J classes in total). The derivative of SoftMax with respect to x_i is as follows:

$$\begin{aligned} \frac{\partial f(x_i)}{\partial x_i} &= \frac{\partial}{\partial x_i}\left(\frac{e^{x_i}}{\sum_j^J e^{x_j}}\right) = \frac{(e^{x_i})' * \sum_j^J e^{x_j} - e^{x_i} * e^{x_i}}{(\sum_j^J e^{x_j})^2} \\ &= \frac{e^{x_i}}{\sum_j^J e^{x_j}} - \frac{e^{x_i}}{\sum_j^J e^{x_j}} * \frac{e^{x_i}}{\sum_j^J e^{x_j}} = f(x_i)(1 - f(x_i)) \end{aligned} \tag{1}$$

  as $f(x_i)$ ranges in $(0, 1)$, the derivative of SoftMax is always positive, thus SoftMax is monotonic.
  Residual function used in ResNet [36] has the property that the mapping from the input to output at each layer before activation function is added with an extra mapping: $H(x) = F(x) + Wx$, where $H(x)$ is the mapping from input to output, $F(x)$ is the residual function. The insight is that it is easier to learn the residual function $F(x)$ than the original mapping $H(x)$. For example, if $H(x)$ is an identity mapping to be learned by the network, it is easier for the network to learn $F(x) = 0$ ($W = 1$) rather than $H(x) = x$. There are different types of residual blocks and we consider the original one in [36]:
  $H(x) = BN(Conv(ReLu(BN(Conv(x))))) + Wx$,
  where Wx is the extra mapping. Assuming a fault occurs at $x_i \in x$, $H(x_i)$ is monotonic if the derivatives of the original mapping $F(x_i)$ and $W_i x_i$ are always positive or negative. However, the derivative of $F(x_i)$ might vary due to fault propagation. For example, assume $H(x) = 100 * ReLu(x - 1) - ReLu(x)$, where one fault propagates into two state spaces (thus there are $x - 1$ and x). The monotonicity of $H(x)$ discontinues according to the value of x: $H(x) = 99x - 100, x > 1$ and $H(x) = -x, x \in (0, 1)$. Therefore, we call the residual block function as approximately monotonic.

Thus we find that almost all the operations in Table 1 exhibit monotonicity. However, this is not an exhaustive list, e.g., there are other activation functions, some of which are non-monotonic (e.g., Swish [61], Sinusoid [28]). Nevertheless, the above models are representative of models used in domains such as object detection (as they yield state-of-art performance and are frequently referred

to by other studies) and the functions in Table 1 are common ML functions. Thus we assume that *the computations in most of the ML models in the application domains mentioned above have the monotonicity property.*

3.4 Error Propagation and (Approximate) Monotonicity

As mentioned earlier, the EP function is a composite function consisting of all the ML functions involved in error propagation. The EP function therefore satisfies the *monotonic* or *approximately monotonic* property, dependent on the ML model.

- **EP with monotonicity:** Recall the example in Fig. 1 (kNN model), where the fault occurs at the tf.add operator, and it eventually affects the distance of the test image to the neighbor image via tf.abs function. In this case, $EP(x) = abs(x)$, which is *monotonic*.
- **EP with approximate monotonicity:** Consider another function in which a single fault x propagates into two state spaces $(x-1, x)$ and ReLu is the subsequent function. The weights associated with the respective data are $(100, -1)$, thus we can model the EP function simply as: $EP(x) = 100 * max(x-1, 0) - max(x, 0)$, which is either monotonically increasing or decreasing in different intervals, e.g., $EP(x) = 99x - 100,\ x > 1$ (monotonically increasing) and $EP(x) = -x, x \in (0, 1)$ (monotonically decreasing). The EP function is thus *approximately monotonic* as its monotonicity during $x > 1$ becomes discontinued when $x \in (0, 1)$, and we approximate that it is monotonic when $x > 0$.

We leverage the (approximate) monotonicity of the EP function for injecting faults. Our methodology applies to EP functions in all ML models, with *either* monotonicity *or* approximate monotonicity, though our approach might incur minor inaccuracy in the latter since it is an approximation. In the following discussion, we use both terms (monotonicity and approximate monotonicity) interchangeably and do not distinguish them if not explicitly specified. EP functions with monotonicity satisfy the following property:

$$|EP(x)| \geq |EP(y)|, (x > y \gg 0) \cup (x < y \ll 0) \quad (2)$$

where x, y are two faults at the bits of both 0 or 1 in the same data, x occurs at high-order bit and y at low-order bit. We consider the faults that deviate considerably from 0 since faults around 0 only yield small deviations, and would not lead to SDCs. The monotonic EP function can be monotonically increasing or decreasing. When EP is monotonically increasing, $EP(x) \leq EP(y) \leq 0, x < y \ll 0$, thus Eq. 2 is satisfied. Monotonically decreasing only occurs in multiply-related functions (e.g., Conv, linear transformation) where the weights are negative. For those functions also, $|f(x)| > |f(y)|, (x > y \gg 0) \cup (x < y \ll 0)$, thus Eq. 2 is satisfied. Hence the EP functions in ML models satisfy Eq. 2. Note that for EP with approximate monotonicity, Eq. 2 might not *always* hold since it is an approximation.

3.5 Binary Fault Injection - *BinFI*

In this section, we discuss how we can leverage the (approximate) monotonicity of the EP functions in ML systems to efficiently pinpoint the critical bits that lead to SDCs. As mentioned earlier, our methodology applies to all ML programs whose EP functions exhibit either monotonicity or approximate monotonicity.

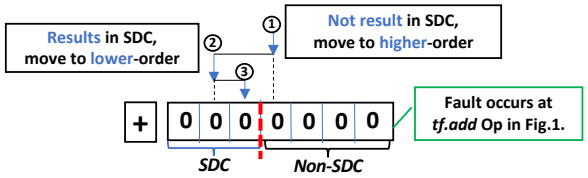

Figure 3: Illustration of binary fault injection. Critical bits are clustered around high-order bits.

With (approximate) monotonicity of the EP function, the outcome by different faults are *sorted* based on the original deviation caused by the faults. Faults at higher-order bits (larger input) would have larger impact on the final outcome (larger output), and are thus more likely to result in SDCs. Therefore, we search for an *SDC-boundary bit*, where *faults at higher-order bits would lead to SDCs and faults from lower-order bits would be masked.* Fig. 3 illustrates this process. Finding such a boundary is similar to searching for a specific target within a sorted array (bits), and thus we decide to use binary-search like algorithm as the FI strategy. We outline the procedure of binFI in Algorithm 1. *getSDCBound* function is the core function to search for the SDC-boundary.

BinFI is run separately for each operator in the model and it considers each data element in the output of the targeted operator. *BinFI* first converts the data into a binary expression (line 2) and then obtains the index of the bits of 0 and 1 respectively (line 3, 4), because faults causing positive and negative deviations would have different impacts (this is why we separate the cases for $x > y \gg 0$ and $x < y \ll 0$). We then find the SDC-boundary bit in the bits of 0 and 1 (line 6, 7).

We illustrate how binFI works on the tf.add operator in the example of Fig. 1 (kNN model). Assuming the fault occurs at $(i, y), y \in [1, 784]$, which corresponds to the nearest neighbor, thus the 0 bits in Fig. 3 are those in the data at (i, y) of the output by the tf.add operator. In this example, whether an SDC will occur depends on whether dis_i is still the nearest neighbor after a fault (Section 3.1).

BinFI first bisects the injection space and injects a fault in the middle bit (line 16, 17). The next injection is based on the result from the current injection . For example, if the fault does not result in an SDC (i.e., dis_i is still the nearest neighbor), the next injection moves to *higher*-order bit (from step 1 to step 2 in Fig. 3), because monotonicity indicates that *no* fault from *lower*-order bits *would* lead to SDCs. More specifically, assume that the result from the fault at step 1 is $dis_i^{'} = dis_i + abs(N)$, where $abs(N)$ is the deviation caused by the *simulated fault*. We can express the results by faults in *lower*-order bits as $dis_i^{''} = dis_i + abs(M)$. According to Eq. 2, $abs(N) > abs(M), N > M > 0$; thus $dis_i^{'} > dis_i^{''}$, where $dis_i^{'}, dis_i^{''}$ are the nearest neighbors of the node.

Moving the next FI to a higher- or lower-order bit is done by adjusting the front or rear index since we are doing binary-search like injection (e.g., line 19 moves the next injection to lower-order bits by adjusting the front index). Step 2 in Fig. 3 shows that the injection causes an SDC, and hence faults from higher-order bits

Algorithm 1: Binary-search fault injection

Data: opOutput ← output from targeted operator
Result: SDC boundary in the bits of 0 and 1, and SDC rate for each element in the targeted operator

1: **for** each **in** opOutput **do**
2: binVal = *binary*(each) // binary conversion
3: 0_list = *getIndexOf_0_bit*(binVal) // build the bit index (bit of 0) from MSB to LSB
4: 1_list = *getIndexOf_1_bit*(binVal)
5: /* results for each element are stored in a list */
6: sdcBound_0.*append*(bound0=*getSDCBound*(binVal, 0_list))
7: sdcBound_1.*append*(bound1=*getSDCBound*(binVal, 1_list))
8: sdcRate.*append*((bound0 + bound1)/*length*(binVal))
9: **end for**
10: **return** sdcBound_1, sdcBound_0, sdcRate
Function: *getSDCBound*(binVal, indexList)
11: /* get the front and rear index for binary splitting */
12: front = 0; // higher-order bit
13: rear = *getLength*(indexList)-1; // lower-order bit
14: sdcBoundary = 0; // by default there is no SDC
15: **while** front ≤ rear **and** front ≠ rear ≠ lastInjectedBit **do**
16: currInjectedBit = indexList[(front + rear)/2]; // binary split
17: FIres = *fault_injection*(binVal, currInjectedBit);
18: **if** FIres results in SDC **then**
19: front = currInjectedBit+1; //move next FI to low-order bit
20: sdcBoundary = currInjectedBit; // index of critical bit
21: **else**
22: rear = currInjectedBit - 1; //move next FI to high-order bit
23: **end if**
24: lastInjectedBit = currInjectedBit
25: **end while**
26: **return** *sdcBoundary*

would also lead to SDCs. The next injection will move to lower-order bits (from step 2 to step 3). We also record the index of the latest bit where a fault could lead to an SDC, and it eventually becomes *sdcBoundary* (line 20). *sdcBound* in line 6, 7 indicates the index of the SDC boundary, as well as how many critical bits (e.g., *sdcBound_1* = 5 means there are 5 critical bits in the bits of 1). Thus we can use them to calculate the SDC rate by calculating the number of critical bits over the total number of bits (line 8).

4 EVALUATION

As mentioned in Section 2.1, we use an open-source FI tool developed in our group called TensorFI [4], for performing FI experiments on TensorFlow-supported ML programs. We modify TensorFI to (1) provide support for DNNs, and (2) support *BinFI*'s approach for injecting faults. The former is necessary as the current version of TensorFI does not have support for complex operations such as convolutions used in DNNs - we added this support and made it capable of injecting faults into DNNs (these modifications have since been merged with the TensorFI mainline tool). The latter is necessary so that we have a uniform baseline to compare *BinFI* with.

We have also made *BinFI* publicly available [5]. For brevity, we refer to the implementation of TensorFI with the above modifications as *BinFI* in the rest of this paper.

We evaluate *BinFI* by asking three research questions as follows:

RQ1: Among all the critical bits in a program, how many of them can be identified by *BinFI*, compared with random FI?

RQ2: How close is the overall SDC rate measured by *BinFI* to the ground truth SDC, compared with that measured by random FI?

RQ3: What is the overhead for *BinFI*, compared with exhaustive and random FI approaches, and how does it vary by data type?

4.1 Experimental Setup

Hardware. All of our experiments were conducted on nodes running Red Hat Enterprise Linux Server 6.4, with Intel Xeon 2.50GHz processors with 12 cores, 64GB memory. We also use a Linux desktop running Ubuntu 16.04 with an Intel i7-4930K 3.40GHz processor with 6 cores, 16GB memory and Nvidia GeForce GT610 GPU. Note that we measure the performance overheads in our experiments in terms of ratios, so the exact hardware configuration does not matter (i.e., we can leverage more HPC resources to accelerate the experiments both for our technique and exhaustive FI, as FI is an embarrassingly parallel problem).

ML models and test datasets. We consider 8 ML models in our evaluation, ranging from simple models, e.g., neural network - NN, kNN, to DNNs that can be used in the self-driving car domains, e.g., Nvidia DAVE systems [19], Comma.ai steering model [5].

We use 6 different datasets including general image classification datasets (Mnist, Cifar-10, ImageNet). These are used for the standard ML models. In addition, we use two datasets to represent two different ML tasks in AVs: *motion planning* and *object detection*. The first dataset is a real-world driving dataset that contains images captured by a camera mounted behind the windshield of a car [6]. The dataset is recorded around Rancho Palos Verdes and San Pedro California, and labeled with steering angles. The second one is the German traffic sign dataset, which contains real-world traffic sign images [39]. We use two different steering models for AVs from: (1) Comma.ai, which is a company developing AV technology and provides several open-source frameworks for AVs such as openpilot [5]; (2) Nvidia DAVE self-driving system [19], which has been implemented in a real car for road tests [11] and has been used as a benchmark in other studies of self-driving cars [59, 78]. We build a VGG11 model [72] to run on the traffic sign dataset. Table 2 summarizes the ML models and test datasets used in this study.

FI campaigns. We perform different FI campaigns on different operations in the network. Due to the time-consuming nature of FI experiments (especially considering that we need to perform exhaustive FI to obtain the ground truth), we decide to evaluate 10 inputs for each benchmark. We also made sure that the inputs were *correctly* classified by the network in the absence of faults. In the case of the driving frame dataset, there was no classification, so we

[4] https://github.com/DependableSystemsLab/TensorFI

[5] https://github.com/DependableSystemsLab/TensorFI-BinaryFI

Table 2: ML models and datasets for evaluation

Dataset	Dataset Description	ML models
MNIST [9]	Hand-written digits	2-layer NN LeNet-4 [48]
Survive [13]	Prediction of patient survival	kNN
Cifar-10 [4]	General images	AlexNet [47]
ImageNet [24]	General images	VGG16 [72]
German traffic sign [39]	Traffic sign images	VGG11 [72]
Driving [6]	Driving video frames	Nvidia Dave [19] Comma.ai [5]

checked if the steering angle was correctly predicted in the absence of faults.

We evaluate all of the operations in the following networks: 2-layer NN, kNN, LeNet-4 and AlexNet (without LRN). However, FI experiments on all operators in the other networks are very time-consuming, and hence we choose for injection one operator *per type* in these networks, e.g., if there are multiple convolution operations, we choose one of them. We perform injections on all the data within the chosen operator except VGG16, which consists of over 3 million data element in one operation's output. Therefore, it is impractical to run FI on all of those data (it will take more than 276525 hours to do exhaustive FI on one operator for one input). So we decide to evaluate our approach on the first 100 data items in the VGG16 model (this took around 13 hours for one input in one operator). We use 32-bit fixed-point data type (1 sign bit, 21 integer bits and 10 mantissa bits), as the fixed-point datatype is more energy efficient than the floating point datatype [22, 32].

4.2 Results

We organize our results by the research questions (RQs).

RQ1: Identifying Critical Bits. To answer this RQ, we consider four safety-critical ML systems used in AVs. This is because SDCs in these systems would be potential safety violations, and hence finding such critical bits that lead to unsafe scenarios is important. We use the steering systems from Nvidia [19] and Comma.ai [5] (on a real-world driving dataset); VGG11 [72] (on a real-world traffic sign dataset) and VGG16 [72] (vehicle images in ImageNet). For the latter two datasets, we consider SDC as any misclassification produced by the system. However, for the two steering models, the output is the steering angle, which is a continuous value, and there is no clear definition of what constitutes an SDC (to our knowledge). Consequently, we came up with three different values of the acceptable threshold for deviations of the steering angle from the correct angle to classify SDCs, namely 5, 30 and 60 degrees. Note that these values include both positive and negative deviations. Fig. 4 shows the test images in our evaluation and exemplify the effect of SDCs. There are two types of deviation since the threshold is for both positive and negative deviations.

Apart from steering the vehicles, it is also crucial for the AVs to correctly identify traffic signs and surrounding vehicles, as incorrect classification in such scenarios could lead to fatal consequences (e.g., misclassifying a "stop" sign as "go ahead" sign). We show in Fig. 5 the potential effects of SDC in such scenarios.

Figure 4: Single-bit flip outcome on the self-controlled steering systems. Blue arrows point to the expected steering angles and red arrows are faulty outputs by the systems.

Figure 5: Single-bit flip outcome on the classifier models. Images at the first row are the input images, those at the second row are the faulty outputs.

We first perform exhaustive FI and record the results from flipping *every* bit in the injection space. We also record all critical bits identified by *BinFI*, and random FI for different numbers of trials. We report the recall (i.e., how many critical bits were found) by *BinFI* and random FI. Exhaustive FI is the baseline and it has a 100% recall as it covers the entire injection space. Fig. 6 presents the recall for different FI approaches on the four safety-critical ML systems. In the interest of space, we report only the recalls for the four simple models: LeNet - 99.84%, AlexNet - 99.61%, kNN and NN - both 100%. We also found that the same operation (e.g., Conv operation) in different layers of a DNN exhibited different resilience (which is in line with the finding in prior work [49]), *BinFI* is able to achieve high recall nonetheless. The reason why *BinFI* has a recall of 100% in kNN and NN is that the EP functions in these models can be modeled as monotonic functions, unlike the other models where the EP function is only approximately monotonic.

Recall is computed as the number of critical bits (found by each FI approach) in the whole state space divided by the total number of critical bits found by exhaustive FI (ground truth). We also provide the number of critical bits (obtained from exhaustive FI) found in each benchmark and the total injection space in Table 3, across all 10 inputs. The critical bits found in the two steering models decrease along with the larger SDC threshold, since larger threshold implies fewer SDCs are flagged.

For the *two steering models* (the left six histograms in Fig. 6), *BinFI* is able to find over 98.71% of critical bits (an average of 99.61%) that would lead to safety violations across all three thresholds.

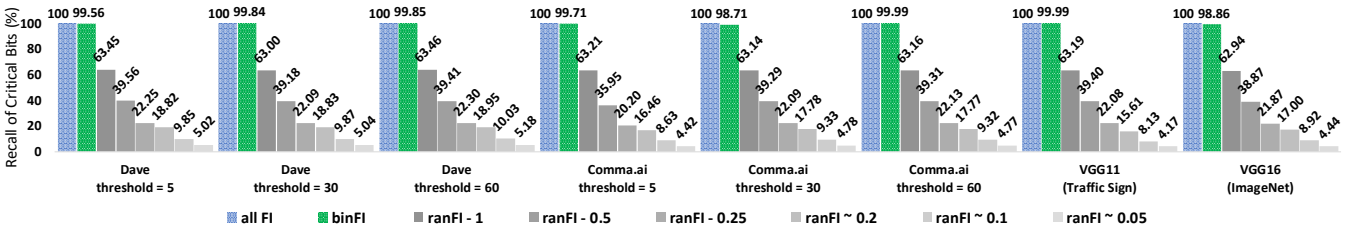

Figure 6: Recall of critical bits by different FI techniques on four safety-critical ML systems. $ranFI-1$ is random FI whose FI trial is the *same* as that by $allFI$; FI trial for $ranFI-0.5$ is *half* of that by $allFI$. $ranFI \sim 0.2$ takes the same FI trials as $BinFI$.

Table 3: Number of critical bits in each benchmark.

Model & Dataset	Critical Bits	Total Bits in FI Space	Percentage (%)
Dave-5 - Driving	2,267,185	4,289,160	52.86
Dave-30 - Driving	2,092,200	4,289,160	48.78
Dave-60 - Driving	1,754,487	4,289,160	40.91
Comma.ai-5 - Driving	3,352,848	8,144,320	41.71
Comma.ai-30 - Driving	2,679,756	8,144,320	32.90
Comma.ai-60 - Driving	2,217,353	8,144,320	27.23
VGG11 - Traffic sign	808,999	4,483,840	18.04
VGG16 - Vehicles	16,919	186,000	9.10

The consistently high recall of *BinFI* when using different SDC thresholds suggest that *BinFI* is *agnostic* to the specific threshold used for classifying SDCs in these models. Similarly, for the other two models, *BinFI* also achieves very high recalls. We thus observe a *consistent* trend across all the benchmarks, each of which exhibits different sensitivity to transient faults as shown in Table 3.

The coverage of random FI depends on the number of trials performed. We consider different numbers of trials for randomFI ranging from 5% to 100% of the trials for the exhaustive FI case. These are labeled with the fraction of trials performed. For example, $ranFI-0.5$, means that the number of trials is 50% that of exhaustive injection. *BinFI* performs about 20% of the trials as exhaustive injection (see Fig. 7 in RQ3), so the corresponding randomFI experiment ($ranFI \sim 0.2$) with the same number of trials identifies about 19% of the critical bits. Even in the best case where the number of trials of random FI is *the same as* that of exhaustive FI, i.e., $ranFI-1.0$, the recall is less than 65%, which is much lower than *BinFI*'s recall of nearly 99%. This is because the bits chosen by random FI are not necessarily unique, especially as the number of trials increases, and hence not all bits are found by random FI.

In addition to recall, we also measure the *precision* (i.e., how many bits identified as critical are indeed critical bits). This is because low precision might raise false alarms and waste unnecessary resources on over-protecting non-critical bits. We report the precision in the four safety-critical models, in Table 4 (precision values for the remaining four models range from 99.31% to 100%). We find that *BinFI* has precision of over 99% across all of the benchmarks, which means that it finds very few unnecessary critical bits.

In summary, we find that BinFI *can achieve an average recall of 99.56% with 99.63% precision, thus demonstrating its efficacy at finding critical bits compared to random FI, which only finds 19% of the critical bits with the same number of trials.*

Table 4: Precision for *BinFI* on identifying critical bits.

Model	DAVE [1] (Driving)	Comma.ai [1] (Driving)	VGG11 (Traffic sign)	VGG16 (Vehicles)
Precision (%)	99.60	99.70	99.99	99.14

[1] Results are averaged from using three thresholds.

RQ2: Overall SDC Evaluation. We calculate the overall SDC probabilities measured by *BinFI* and random FI for the ML models. Note that the overall SDC probability is a product of the number of bits in the operator, as well as the SDC probability per bit. So we weight the per bit SDC with the number of bits in the operator to obtain the overall SDC probability. For example, the overall SDC probability for an operator with 100 bits (with SDC probability 20%) and another operator with 10000 bits (with SDC probability 5%) will be $(100 * 0.20 + 10000 * 0.05)/(100 + 10000) = 5.15\%$.

To quantify the accuracy of *BinFI* in measuring the overall SDC, we measure the deviation of the SDC probabilities from the ground truth obtained through exhaustive FI in Table 5 (error bars at the 95% confidence intervals are shown below the SDC rates in the table). We limit the number of trials performed by random FI to those performed by *BinFI* to obtain a fair comparison. Overall, we find that the SDC probabilities measured by *BinFI* are very close to the ground truth obtained through exhaustive FI. Table 5 also shows that *BinFI* achieves 0% deviation in two ML models (NN and kNN), which is because the EP function in these two models are monotonic.

While the above results demonstrate that *BinFI* can be used to obtain accurate estimates of the overall resilience, random FI achieves nearly the same results with a much lower number of trials. Therefore, if the goal is to only obtain the overall SDC probabilities, then random FI is more efficient than *BinFI*.

RQ3: Performance Overhead. We evaluate the performance overhead for each FI technique in terms of the number of FI trials, as the absolute times are machine-dependent. We show the results for AlexNet and VGG11, in Fig. 7. The overall FI trials for VGG11 are lower than those for AlexNet as we do not inject fault into all operators in VGG11. Fig. 7 shows that the number of FI trials performed by *BinFI* is around 20% of that by exhaustive FI. This is expected as *BinFI* performs a binary-search and we use a 32-bit datatype - $log_2 31 \approx 5$. Note that the value is not $5/31 \approx 16\%$ since

Table 5: Overall SDC deviation compared with ground truth. Deviation shown in percentage (%), and error bars shown at the 95% confidence intervals for random FI.

Model	*BinFI*	ranFI∼0.2	ranFI∼0.1	ranFI∼0.05
Dave-5 (Driving)	0.070	0.100 (±0.01 ∼ ±0.31)	0.101 (±0.02 ∼ ±0.44)	0.119 (±0.02 ∼ ±0.62)
Dave-30 (Driving)	0.036	0.096 (±0.04 ∼ ±0.30)	0.134 (±0.06 ∼ ±0.42)	0.172 (±0.09 ∼ ±0.59)
Dave-60 (Driving)	0.125	0.116 (±0.05 ∼ ±0.29)	0.118 (±0.08 ∼ ±0.42)	0.234 (±0.11 ∼ ±0.59)
Comma.ai-5 (Driving)	0.049	0.064 (±0.23 ∼ ±0.24)	0.040 (±0.33 ∼ ±0.34)	0.095 (±0.47 ∼ ±0.48)
Comma.ai-30 (Driving)	0.212	0.092 (±0.22 ∼ ±0.24)	0.160 (±0.31 ∼ ±0.34)	0.206 (±0.45 ∼ ±0.48)
Comma.ai-60 (Driving)	0.008	0.060 (±0.21 ∼ ±0.22)	0.094 (±0.30 ∼ ±0.31)	0.212 (±0.43 ∼ ±0.44)
VGG11 (Traffic sign)	0.002	0.101 (±0.23 ∼ ±0.26)	0.156 (±0.33 ∼ ±0.38)	0.199 (±0.47 ∼ ±0.53)
VGG16 (ImageNet)	0.042	1.039 (±0.62 ∼ ±1.07)	0.778 (±0.86 ∼ ±1.58)	0.800 (±1.28 ∼ ±2.14)
AlexNet (Cifar-10)	0.068	0.319 (±0.15 ∼ ±0.18)	0.420 (±0.22 ∼ ±0.25)	0.585 (±0.31 ∼ ±0.36)
LeNet (Mnist)	0.030	0.228 (±0.45 ∼ ±0.46)	0.366 (±0.64 ∼ ±0.65)	3.38 (±0.887 ∼ ±0.95)
NN (Mnist)	**0.000**	0.849 (±0.33 ∼ ±1.1)	0.930 (±0.47 ∼ ±1.55)	0.989 (±0.67 ∼ ±2.20)
kNN (Survival)	**0.000**	0.196 (±0.05 ∼ ±0.2)	0.190 (±0.19 ∼ ±0.32)	0.197 (±0.195 ∼ ±0.47)

we separately do injection for 0-bits and 1-bits. Thus, it is closer to 6/31(≈ 20%). We observe a similar trend in all the benchmarks (not shown due to space constraints).

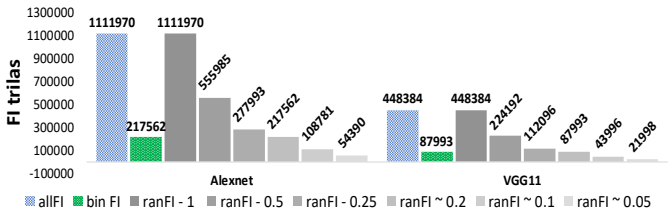

Figure 7: FI trials (overhead) for different FI techniques to identify critical bits in AlexNet and VGG11.

The performance overhead of *BinFI* also depends on the number of bits used in the data representation. In general, the overhead gains increase as the number of bits increases (as *BinFI*'s time grows logarithmically with the number of bits, while exhaustive FI's time grows linearly with the number of bits). To validate this intuition, we evaluate *BinFI* on VGG11 using datatypes with different width (16-bit, 32-bit and 64-bit) and compare its overhead with that of exhaustive FI in Fig. 8. We find that the growth rate of *BinFI* is indeed logarithmic with the number of bits. We also measure the recall and precision of *BinFI* for the three data-types. The recall values are 100%, 99.97%, 99.99% for 16-bit, 32-bit and 64 bits respectively, while the respective precision values are 100%, 99.96%, 99.94%. This shows that the precision and recall values of *BinFI* are independent of the datatype.

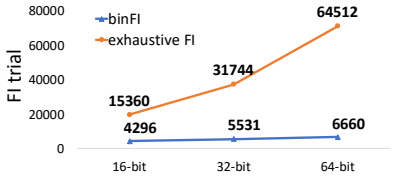

Figure 8: Numbers of FI trials by *BinFI* and exhaustive FI to identify critical bits in VGG11 with different datatypes.

5 DISCUSSION

We start this section by discussing the inaccuracy of *BinFI*, followed by the effects of non-monotonicity on its efficacy. Finally, we reflect on the implications of *BinFI*, and its application to other HPC areas.

5.1 Inaccuracy of *BinFI*

As mentioned in Section 3.4, the EP function is often only approximately monotonic - this is the main source of inaccuracy for *BinFI*, as it may overlook the critical bits in the non-monotonic portions of the function. Assuming $EP(x) = 2 * max(x-1,0) - max(x,0)$ and a fault raises a deviation of $x = 2, EP(2) = 0$, which does not result in an SDC. According to Eq. 2, *BinFI* regards that *all* the faults from $0 < y < 2$ will not cause SDCs as $|Eq(y)| \leq 0$. However, $|EP(0.5)| = 0.5$, which violates Eq. 2. A fault incurring a deviation of 0.5 could be a critical bit unidentified by *BinFI* (thus resulting in inaccuracy). This is the reason why *BinFI* incurs minor inaccuracy except for two models in Table 5, in which the EP function is monotonic (not just approximately so). Nevertheless, our evaluation shows that *BinFI* incurs only minor inaccuracy as it has an average recall of 99.56% with 99.63% precision, and the overall SDC rate is very close to ground truth as well. This is because the non-monotonicity occurs in most cases when the fault is small in magnitude, and is hence unlikely to lead to an SDC.

5.2 Effect of Non-Monotonicity

While our discussion in Section 3.2 shows that many computations within the state-of-art ML models are monotonic, there are also some models that use non-monotonic functions. Though *BinFI* requires the functions to be monotonic so that the EP function is (approximately) monotonic, we also want to measure the effects when *BinFI* is run on those models using functions that are not monotonic. Therefore, we conduct an experiment to evaluate *BinFI* on two networks using different non-monotonic functions. Specifically, we use a Neural Network using a Swish activation function [61], and AlexNet with LRN [47]. As before, we measure the recall and precision of *BinFI* on these models.

For the NN model, Fig. 9 shows that *BinFI* has a recall of 98.9% and a precision of 97.3%, which is quite high. The main reason is that Swish function is monotonic across a non-trivial interval, thus exhibiting approximate monotonicity, so *BinFI* works well on it. On the other hand, when it comes to AlexNet with LRN, *BinFI* has high recall but low precision, which means *BinFI* is not suitable for this model as LRN is non-monotonic in nature (Section 3.3).

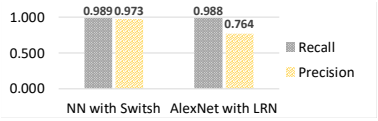

Figure 9: Recall and precision for *BinFI* on ML models with non-monotonic functions.

5.3 Implications for Protection

Knowing the critical bits in ML systems can lead to the adoption of configurable fault-tolerance techniques with low costs, e.g., [45, 46, 52, 82], without resorting to conservative approaches [40, 49, 51]. For example, Li et al. [52] enable dynamic fault tolerance at runtime, given that they know when the protection is needed to save energy. Similarly, Krause et al. [46] propose a data-driven voltage over-scaling technique, where the supply voltage can be dynamically increased if the errors exceed a given threshold in a certain region of code. Given that *BinFI* is able to identify the critical bits, one can adopt these approaches to dynamically protect the ML systems with low costs. Moreover, *BinFI* can also identify the parts of application that might be exploited by malicious attackers, hence guiding the security protection against hardware fault attacks towards ML systems [38].

However, while *BinFI* is able to identify 99.56% of the critical bits in an application, its coverage is not 100%. This means that there may be a few critical bits that are left unidentified when using *BinFI*. This is a limitation of our approach's assumption of monotonicity. Unfortunately, there is no easy remedy for this problem, as identifying 100% of the bits requires exhaustive FI, which is highly time-consuming. Our study is a first step towards low-cost protection on safety-critical ML applications and we believe that missing a small proportion of the critical bits is an acceptable tradeoff given: (1) the significant cost savings in *BinFI*; (2) the relatively rare occurence of soft errors; (3) not all critical bits would result in actual failures in the systems [43].

5.4 Application to Other Areas of HPC

In this paper, we mainly use *BinFI* to evaluate safety-critical MLs in the AVs domain, which is an emerging example of ML in HPC. However, *BinFI* is not confined to this domain and there are many other areas of application of *BinFI* in the HPC context. We consider 3 examples below. (1) Xiong et al. [80] use thousands of GPU cores to implement a DNN for the detection of atrial fibrillation. (2) Yoon et al. [81] use DNNs to extract information from cancer pathology reports in cancer registries, using supercomputer facility. (3) Cong et al. [21] use a cluster of GPUs to accelerate ML tasks for action recognition, which exhibits better performance in terms of both speedup and accuracy . There are many other examples of ML used in safety-critical domains [26, 44, 60]. In these domains, transient faults can have serious implications on safety, and hence *BinFI* can be used to find the safety-critical bits in the application. We defer exploration of ML in other safety critical domains to future work.

In Section 3.3 we discuss our observation of monotonicity found in common ML functions, based on which we design *BinFI* to efficiently identify safety-critical bits in ML programs. While it is possible that *BinFI* can be applied in other application domains if the computations in the application exhibit the monotonicity property, we believe this is unlikely to be the case in general purpose programs. This is because different faults in these programs could lead to the execution of different program paths [33, 51, 58]. For example, a larger fault could cause a different program path to be executed, whose output is not necessarily larger than the output from another program path caused by a smaller fault (e.g., due to the different computations performed), thereby violating monotonicity. We defer the exploration of non-ML applications to future work.

6 RELATED WORK

We classify related work into three broad areas.

Testing of ML: Pei et al. [59] propose a novel approach to generate corner case inputs to trigger unexpected behaviors of the model. They analyze the decision boundary of different ML models and leverage gradient ascent to modify the image to efficiently trigger differential behaviors in different DNNs. Tian et al. [78] use transformation matrices (e.g., scale, rotate the image) to automatically generate corner case images. Ma et. al. [54] design a mutation framework to mutate the ML model, which is then used to evaluate the quality of the test data. The idea of mutating the ML model [54] is similar to fault injection. However, unlike our work which injects transient faults, the faults they inject have to do with emulating software faults such as removing a neuron or shuffling the data. Rubaiyat et al. [67] build a strategic software FI framework, which leverages hazard analysis to identify potential unsafe scenarios to prune the injection space. However, their approach suffers from low error coverage (less than 36%). None of these approaches consider hardware transient faults, which are growing in frequency and can trigger undesirable consequences in ML systems. Moreover, none of the above approaches leverage monotonicity of the models' functions to perform efficient fault injection.

Error resilience of ML: Li et al. [49] build a fault injector to randomly inject transient hardware faults in ML application running on specialized hardware accelerators. Using the injector, they study the resilience of the program under different conditions by varying different system parameters such as data types. A recent study designs a DNN-specific FI framework to inject faults into real hardware and studies the trade off between the model accuracy and the fault rate [62]. Santos et al. [25] investigate the resilience of ML under mixed-precision architectures by conducting neutron beam experiments. These papers measure the overall resilience of ML systems using random FI, while our approach identifies the bits that can lead to safety violations in these systems. As we showed in this paper, random FI achieves very poor coverage for identifying the critical bits in ML systems.

Accelerating fault injection: Given the high overhead of FI experiments, numerous studies have proposed to accelerate the FI experiments by pruning the FI space or even predicting the error resilience without performing FI [30, 33, 51, 69]. Hari et al. [33] propose Relyzer, a FI technique that exploits fault equivalence (an observation that faults that propagate in similar paths are likely to result in similar outputs) to selectively perform FI on the pilot instructions that are representative of fault propagation. In follow up work, the authors propose GangES, which finds that faults resulting in the same intermediate execution state will produce the same

faulty output [69], thus pruning the FI space further. Li et al. [51] propose Trident, a framework that can predict the SDC rate of the instructions without performing any FI. The key insight is that they can model the error propagation probability in data-dependency, control-flow and memory levels jointly to predict the SDC rate. They find that the Trident model is able to predict both the overall SDC rate of the program and those of individual instructions. However, none of these studies are tailored for ML programs and their focus is on measuring the overall resilience of the system, unlike *BinFI* that can identify the critical bits.

7 CONCLUSION

In this work, we propose an efficient fault injector to identify the critical bits in ML systems under the presence of hardware transient faults. Our insight is based on the observation that many of the ML computations are monotonic, which constrains their fault propagation behavior. We thus identify the existence of the SDC boundary, where faults from higher-order bits would result in SDCs while faults at lower-order bits would be masked. Finally, we design a binary-search like fault injector to identify the SDC boundary, and implement it as a tool called *BinFI* for ML programs written using the TensorFlow framework.

We evaluate *BinFI* on 8 ML benchmarks including ML systems that can be deployed in AVs. Our evaluation demonstrates that *BinFI* can correctly identify 99.56% of the critical bits with 99.63% precision, which significant outperforms conventional random FI-based approaches. It also incurs significantly lower overhead than exhaustive FI techniques (by 5X). *BinFI* can also accurately measure the overall resilience of the application.

As future work, we plan to (1) extend *BinFI* to other ML frameworks than TensorFlow, (2) consider other safety-critical ML applications than AVs, and (3) explore selective protection techniques based on the results from *BinFI*.

BinFI is publicly available at the following URL: https://github.com/DependableSystemsLab/TensorFI-BinaryFI

ACKNOWLEDGMENTS

This work was funded in part by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery grant and Strategic grant programmes. We thank the anonymous reviewers of SC'19 for their comments which helped improve the paper.

This manuscript has been approved for unlimited release and has been assigned LA-UR-19-27921. This work has been co-authored by an employee of Triad National Security, LLC which operates Los Alamos National Laboratory under Contract No. 89233218CNA000001 with the U.S. Department of Energy/National Nuclear Security Administration. The publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for United States Government purposes.

REFERENCES

[1] Autonomous and ADAS test cars produce over 11 TB of data per day. https://www.tuxera.com/blog/autonomous-and-adas-test-cars-produce-over-11-tb-of-data-per-day/
[2] Autonomous Car - A New Driver for Resilient Computing and Design-for-Test. https://nepp.nasa.gov/workshops/etw2016/talks/15WED/20160615-0930-Autonomous_Saxena-Nirmal-Saxena-Rec2016Jun16-nasaNEPP.pdf
[3] Autumn model in Udacity challenge. https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/autumn
[4] Cifar dataset. https://www.cs.toronto.edu/~kriz/cifar.html
[5] comma.ai's steering model. https://github.com/commaai/research
[6] Driving dataset. https://github.com/SullyChen/driving-datasets
[7] Epoch model in Udacity challenge. https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/cg23
[8] Functional Safety Methodologies for Automotive Applications. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/solutions/automotive-functional-safety-wp.pdf
[9] Mnist dataset. http://yann.lecun.com/exdb/mnist/
[10] NVIDIA DRIVE AGX. https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/
[11] On-road tests for Nvidia Dave system. https://devblogs.nvidia.com/deep-learning-self-driving-cars/
[12] Rambo. https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/rambo
[13] Survival dataset. https://archive.ics.uci.edu/ml/datasets/Haberman‘s+Survival
[14] Tensorflow Popularity. https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a
[15] Training AI for Self-Driving Vehicles: the Challenge of Scale. https://devblogs.nvidia.com/training-self-driving-vehicles-challenge-scale/
[16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.
[17] Rizwan A Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F DeMara, Chen-Yong Cher, and Pradip Bose. 2015. Understanding the propagation of transient errors in HPC applications. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.
[18] Subho S Banerjee, Saurabh Jha, James Cyriac, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2018. Hands Off the Wheel in Autonomous Vehicles?: A Systems Perspective on over a Million Miles of Field Data. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 586–597.
[19] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).
[20] Chun-Kai Chang, Sangkug Lym, Nicholas Kelly, Michael B Sullivan, and Mattan Erez. 2018. Evaluating and accelerating high-fidelity error injection for HPC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 45.
[21] G Cong, G Domeniconi, J Shapiro, F Zhou, and BY Chen. 2018. *Accelerating Deep Neural Network Training for Action Recognition on a Cluster of GPUs*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).
[22] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).
[23] Nathan DeBardeleben, James Laros, John T Daly, Stephen L Scott, Christian Engelmann, and Bill Harrod. 2009. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. *Whitepaper, Dec* (2009).
[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. (2009).
[25] Fernando Fernandes dos Santos, Caio Lunardi, Daniel Oliveira, Fabiano Libano, and Paolo Rech. 2019. Reliability Evaluation of Mixed-Precision Architectures. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 238–249.
[26] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* 542, 7639 (2017), 115.
[27] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. 2014. Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 221–230.
[28] Michael S Gashler and Stephen C Ashmore. 2014. Training deep fourier neural networks to fit time-series data. In *International Conference on Intelligent*

Computing. Springer, 48–55.

[29] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S Nikolopoulos, and Martin Schulz. 2017. Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 29.

[30] Jason George, Bo Marr, Bilge ES Akgul, and Krishna V Palem. 2006. Probabilistic arithmetic and energy efficient embedded signal processing. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 158–168.

[31] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 739–743.

[32] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*. 1737–1746.

[33] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 123–134.

[34] Simon Haykin. 1994. *Neural networks*. Vol. 2. Prentice hall New York.

[35] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at Facebook: a datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 620–629.

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

[37] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[38] 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. *arXiv preprint arXiv:1906.01017* (2019).

[39] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. 2013. Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In *The 2013 international joint conference on neural networks (IJCNN)*. IEEE, 1–8.

[40] Jie S Hu, Feihui Li, Vijay Degalahal, Mahmut Kandemir, Narayanan Vijaykrishnan, and Mary J Irwin. 2005. Compiler-directed instruction duplication for soft error detection. In *Design, Automation and Test in Europe*. IEEE, 1056–1057.

[41] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[42] Saurabh Jha, Subho S Banerjee, James Cyriac, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2018. Avfi: Fault injection for autonomous vehicles. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 55–56.

[43] Saurabh Jha, Timothy Tsai, Subho Banerjee, Siva Kumar Sastry Hari, Michael Sullivan, Steve Keckler, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2019. ML-based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[44] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. 2016. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE, 1–10.

[45] Zvi M Kedem, Vincent J Mooney, Kirthi Krishna Muntimadugu, and Krishna V Palem. 2011. An approach to energy-error tradeoffs in approximate ripple carry adders. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*. IEEE Press, 211–216.

[46] Philipp Klaus Krause and Ilia Polian. 2011. Adaptive voltage over-scaling for resilient applications. In *2011 Design, Automation & Test in Europe*. IEEE, 1–6.

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

[48] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. 1990. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*. 396–404.

[49] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W Keckler. 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*. ACM, 8.

[50] Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. 2018. TensorFI: A Configurable Fault Injector for TensorFlow Applications. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 313–320.

[51] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. 2018. Modeling soft-error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 27–38.

[52] Wenchao Li, Susmit Jha, and Sanjit A Seshia. 2013. Generating control logic for optimized soft error resilience. In *Proceedings of the 9th Workshop on Silicon Errors in Logic-System Effects (SELSEâĂŹ13), Palo Alto, CA, USA*. Citeseer.

[53] Robert E Lyons and Wouter Vanderkulk. 1962. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development* 6, 2 (1962), 200–209.

[54] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 100–111.

[55] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. 2013. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, Vol. 30. 3.

[56] Marisol Monterrubio-Velasco, José Carlos Carrasco-Jimenez, Octavio Castillo-Reyes, Fernando Cucchietti, and Josep De la Puente. 2018. A Machine Learning Approach for Parameter Screening in Earthquake Simulation. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 348–355.

[57] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.

[58] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. 2002. Control-flow checking by software signatures. *IEEE transactions on Reliability* 51, 1 (2002), 111–122.

[59] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 1–18.

[60] Pranav Rajpurkar, Awni Y Hannun, Masoumeh Haghpanahi, Codie Bourn, and Andrew Y Ng. 2017. Cardiologist-level arrhythmia detection with convolutional neural networks. *arXiv preprint arXiv:1707.01836* (2017).

[61] Prajit Ramachandran, Barret Zoph, and Quoc V Le. 2017. Searching for activation functions. *arXiv preprint arXiv:1710.05941* (2017).

[62] 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 *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[63] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[64] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7263–7271.

[65] Daniel A Reed and Jack Dongarra. 2015. Exascale computing and big data. *Commun. ACM* 58, 7 (2015), 56–68.

[66] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*. 91–99.

[67] Abu Hasnat Mohammad Rubaiyat, Yongming Qin, and Homa Alemzadeh. 2018. Experimental resilience assessment of an open-source driving agent. *arXiv preprint arXiv:1807.06172* (2018).

[68] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. 2017. 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, 97–108.

[69] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V Adve, and Helia Naeimi. 2014. GangES: Gang error simulation for hardware resiliency evaluation. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 61–72.

[70] Bianca Schroeder and Garth A Gibson. 2007. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, Vol. 78. IOP Publishing, 012022.

[71] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484.

[72] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[73] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. 2014. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications* 28, 2 (2014), 129–173.

[74] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[75] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[76] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[77] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[78] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. ACM, 303–314.

[79] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. 2014. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 375–382.

[80] Zhaohan Xiong, Martin K Stiles, and Jichao Zhao. 2017. Robust ECG signal classification for detection of atrial fibrillation using a novel neural network. In *2017 Computing in Cardiology (CinC)*. IEEE, 1–4.

[81] Hong-Jun Yoon, Arvind Ramanathan, and Georgia Tourassi. 2016. Multi-task deep neural networks for automated extraction of primary site and laterality information from cancer pathology reports. In *INNS Conference on Big Data*. Springer, 195–204.

[82] Ming Zhang, Subhasish Mitra, TM Mak, Norbert Seifert, Nicholas J Wang, Quan Shi, Kee Sup Kim, Naresh R Shanbhag, and Sanjay J Patel. 2006. Sequential element design with built-in soft error resilience. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14, 12 (2006), 1368–1378.