LLTFI: Framework Agnostic Fault Injection for Machine Learning Applications (Tools and Artifact Track)

Udit Kumar Agarwal, Abraham Chan, Karthik Pattabiraman
The University of British Columbia , Canada.
Email: uditag97@student.ubc.ca, abrahamc@ece.ubc.ca, karthikp@ece.ubc.ca

2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE) | 978-1-6654-5132-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISSRE55969.2022.00036

***Abstract*—As machine learning (ML) has become more prevalent across many critical domains, so has the need to understand ML applications' resilience. While prior work like TensorFI [1], MindFI [2], and PyTorchFI [3] has focused on building ML fault injectors for specific ML frameworks, there has been little work on performing fault injection (FI) for ML applications written in multiple frameworks. We present LLTFI, a framework-agnostic fault injection tool for ML applications, allowing users to run FI experiments on ML applications at the LLVM IR level. LLTFI provides users with finer FI granularity at the level of instructions, and a better understanding of how faults manifest and propagate between different ML components. We evaluate LLTFI on six ML programs and compare it with TensorFI. We found significant differences in the Silent Data Corruption (SDC) rates for similar faults between the two tools. Finally, we use LLTFI to evaluate the efficacy of selective instruction duplication - an error mitigation technique - for ML programs.**

***Index Terms*—Error resilience, Machine learning, Testing**

I. Introduction

Machine learning (ML) applications have been ubiquitously deployed across critical domains such as autonomous vehicles (AVs), and medical diagnosis. For example, AVs that make incorrect decisions can cause injuries and fatalities. Therefore, it is important to understand the resilience of ML applications.

While faults can arise due to software (e.g., software bugs) or hardware (e.g., radiation-induced bit flips), we focus on the latter, and specifically on transient hardware faults. Due to their unpredictable nature, transient hardware faults are difficult to avoid and hence need mitigation. Fault Injection (FI) is the traditional way to evaluate the resilience of systems to hardware faults. Software-implemented fault injection (SWiFI) has become the preferred choice compared to hardware injection due to its cost and time efficiency benefits [4].

Most ML developers use popular ML frameworks such as TensorFlow [5] and PyTorch [6]. Therefore, ML-specific SWiFI tools have recently been developed e.g., TensorFI [1], PyTorchFI [3], and TF-DM [7] - we refer to these as *ML FI tools.* ML FI tools inject faults directly into tensor operators and tensors, representing weights and neurons in the ML application. However, unlike low-level SWiFI tools, ML FI tools are unable to inject faults into specific registers and instructions. For example, in these frameworks, a convolution operation can be lowered into many low-level instructions such as add, multiply, and arithmetic shifts. A hardware fault may only affect a single instruction (i.e., a left shift), but an ML FI tool would modify the output of the entire convolution operation due to its limited visibility.

Further, complex ML-based systems for autonomous vehicles such as Baidu's Apollo [8] and Comma.ai's openpilot [9] incorporate ML models, developed over multiple ML frameworks. For instance, Baidu Apollo contains at least 28 different ML models, written in frameworks including TensorFlow and PyTorch, along with programmed components in C++ [10]. Therefore, it is important for a single FI tool to support multiple ML frameworks in a consistent manner.

In this paper, we introduce LLTFI[1] , a heterogeneous SWiFI tool that supports fault injection in both *C/C++ programs and ML applications written using high-level ML frameworks such as TensorFlow and PyTorch*. In contrast to ML FI tools, LLTFI can inject into specific instructions and registers in ML models written in high-level frameworks, and also visualize fault injection in low-level control flow graphs (that correspond to instructions and registers) to study how errors propagate between instructions and nodes in the ML application. LLTFI performs these tasks in a framework-independent manner and can hence work with a wide variety of ML frameworks. *To the best of our knowledge, LLTFI is the only FI tool that enables the injection of hardware faults into ML applications that are written using (most) ML frameworks.*

We develop LLTFI on top of the widely-used and publicly available FI tool, LLFI [12], which performs FI at the LLVM Intermediate Representation (IR) level [13] in C/C++ programs.We choose the LLVM IR abstraction level as it exposes instruction and virtual register information while remaining hardware independent. Further, LLVM IR supports a wide range of target platforms. Recently, LLVM has been extended to support ML platforms - this extension is called MLIR [14]. We leverage another framework called ONNX [15] that provides platform-neutral representations of ML models. ONNX currently supports 24 different ML frameworks, including Tensorflow, Keras, PyTorch, Caffe, etc. Therefore, ML programs written in these any of these ML frameworks can be converted to their ONNX representations. We then combine ONNX and MLIR in LLTFI to provide platform-independent fault

[1]LLTFI stands for Low-Level Tensor Fault Injector, and is publicly available on Github: https://github.com/DependableSystemsLab/LLTFI. This paper extends our previously published workshop paper on LLTFI [11].

2332-6549/22/$31.00 ©2022 IEEE
DOI 10.1109/ISSRE55969.2022.00036

injection of ML models. Our use of LLFI allows LLTFI to inject faults into both ML programs and C/C++ programs.

LLTFI is externally configurable with a YAML file and requires neither any changes to the ML code nor any annotations. Further, it can support a wide range of fault modes and also allow visualization of error propagation. Finally, it can be extended to provide selective protection of ML programs.

We evaluate LLTFI with respect to ML FI frameworks such as TensorFI, in terms of the Silent Data Corruption (SDC) rate (i.e., undetected incorrect output) and performance overhead, for a set of six popular ML applications and three datasets. We find that there are considerable differences between the SDC rates of the two tools for the same faults, *with LLTFI exhibiting significantly lower SDC rates (average: 3.4X lower) than TensorFI.* Further, the runtime overhead of LLTFI is 1.27X lower than that of TensorFI, on average. Finally, we extended LLTFI to implement selective instruction duplication for detecting and mitigating SDCs in ML programs, and measure its runtime overhead and SDC rate reduction.

II. Background and Fault Model

In this section, we explain what ML Frameworks, LLFI, MLIR, ONNX are, and why they are relevant to LLTFI. We also explain LLTFI's fault model.

A. Background

ML Frameworks. ML developers typically use ML frameworks so they can avoid rewriting well optimized ML operators and exploit support for hardware acceleration. TensorFlow [5] and PyTorch [6] are examples of popular ML frameworks. Both frameworks represent sequences of ML operations as directed acyclic graphs to facilitate optimizations.

LLFI [12] is a low level fault injector that injects faults at the LLVM Intermediate Representation (IR) level. It leverages the LLVM [13] compiler to compile programs, written in high level programming languages such as C/C++, to LLVM IR. LLFI instruments (i.e., inserts fault injection calls) into the LLVM IR code. Subsequently, LLFI randomly activates these fault injection calls during runtime to simulate fault injection.

ONNX [15] stands for Open Neural Network Exchange, which is an open serialization format for ML models. It provides a common set of ML operators across popular ML frameworks like TensorFlow and PyTorch, as well as others like scikit-learn. An example ONNX operator is MaxPool, since many ML models utilize maximum pooling. At the time of writing, ONNX covers 91.53% of the common ML operators[2]. Further, the ONNX specification is still evolving with new operators being added with every version. Therefore, we expect that it will cover more ML operators in the future.

MLIR [14] stands for Multi-Level Intermediate Representation. It is a hybrid IR that supports heterogeneous optimizations across ML operations. MLIR has multiple dialects - each represents a group of MLIR operations. In this paper, we refer to the LLVM dialect of MLIR as MLIR. Despite MLIR's LLVM dialect being isomorphic with LLVM IR, it does not have LLVM IR features like phi nodes. We use MLIR as an intermediary between LLVM IR and ONNX.

[2]http://www.xavierdupre.fr/app/onnxcustom/helpsphinx/onnxmd/onnx_docs/TestCoverage.html#summary

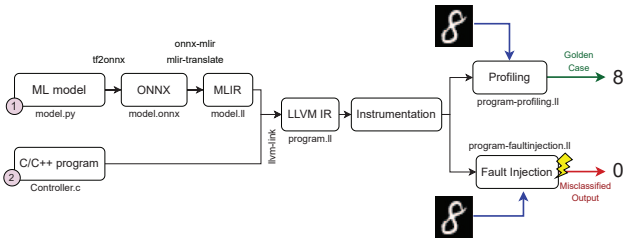


Fig. 1: LLTFI's workflow.

TensorFI [1] is an application-level ML FI tool that is widely used by many FI studies [16, 17]. TensorFI works by injecting bit-flip faults at runtime in the intermediate layer outputs of the ML model. Like other ML FI tools, TensorFI operates at the operator-level of the ML model (e.g., TensorFlow graphs). We use TensorFI as it is representative of framework-specific ML FI tools to contrast against LLTFI.

B. Fault Model

We primarily consider transient hardware faults in our work. Transient hardware faults occur as a result of random particle strikes on flip-flops and the chip's logic elements, and cause bits to flip from 0 to 1 or vice versa [18]. We do not consider faults in the instruction encoding or in the control logic of the processor. We also assume that these faults do not modify the structure of the control flow graph of the ML application. Moreover, although the tool can be used with multiple hardware backends supported by LLVM, we only consider the general CPU hardware model in this work. These assumptions are in line with other papers [1–3, 12, 19, 20].

III. LLTFI Approach

As mentioned, LLTFI is built on top of LLFI [12] and is fully backwards compatible with it. While developing LLTFI, we upgraded the entire LLFI tool, which was originally written for LLVM 3.4, to LLVM 12.0. This is important as LLVM 3.4 has no support for MLIR, which is required to lower ML programs to LLVM IR. LLTFI provides a single script that converts ML models into LLVM IR, for convenience.

We first describe the workflow of LLTFI, followed by our design choices, and finally the fault types supported by LLTFI.

A. Workflow

Fig. 1 shows the workflow diagram of LLTFI for ML models (that use ML Frameworks). Firstly, developers write their models using the ML framework (e.g., TensorFlow, PyTorch), and train their models. The trained models are then exported to a saved model format (like Tensorflow's SavedModel). Secondly, LLTFI converts the saved model to the ONNX [15] format. Thirdly, the ONNX file is converted into MLIR through ONNX-MLIR [21]. Finally, MLIR is converted into LLVM IR, using the `mlir-translate` tool in LLVM 12.0.

To run the ML model, a controller program, written in C or C++ is required. LLTFI directly compiles the controller program into LLVM IR, which can then be linked to the ML program. After this step, LLTFI can inject faults into the LLVM IR of the model at runtime without compilation.

B. Design Choices

Selecting the lowering mechanism. There are multiple methods to lower high-level ML models to intermediate representation (IR). We explain why LLTFI lowers ML models to MLIR using ONNX-MLIR [21] over other alternatives.

Glow is a ML compiler [22], which lowers a neural network dataflow graph into high-level Glow IR, followed by low-level Glow IR. However, Glow does not lower to LLVM IR. XLA [23] and TVM [24] are both compilers for lowering TensorFlow and PyTorch models into LLVM IR. XLA first lowers into XLA high-level optimizer representation, while TVM first lowers into Halide IR [25], before lowering to LLVM IR. While XLA and Halide both enable more optimizations than MLIR, their compilation time overhead is also greater.

Moreover, one of the objectives of this work is to develop a unified fault injector that works with both C/C++ and ML programs (in a framework-agnostic manner). While TVM and XLA can be used for ML programs, they do not work with C/C++ programs. In LLTFI, we used ONNX-MLIR to lower ML models to LLVM IR, followed by fault injection using LLFI, which can also be used for C/C++ applications. Therefore LLTFI is backward compatible with LLFI and can be used with both C/C++ and ML programs.

Adapting low-level fault injection for ML models.

LLTFI injects faults into values chosen at runtime with uniform probability, including address values. Since address values are abundant, and accessing illegal addresses often cause crashes, fault injected models are more likely to crash than to produce silent data corruptions (SDCs) (e.g., misclassifications). This is problematic for comparing the FI results with ML FI tools, which do not typically cause crashes [1].

To avoid the problem of injecting into address values, we provide an LLVM pass for users to skip certain instructions while injecting directly into math and logic operations in the basic blocks representing the high-level ML computation graphs. Additionally, this pass enables targeted FI into selected ML model layers. The primary challenge was to map Tensor operators in the high-level ML computation graph to their respective LLVM IR code blocks. We addressed this challenge in the new pass by utilizing the debug instrumentation provided by the ONNX-MLIR tool to uniquely identify the LLVM IR code block(s) of different Tensor operators.

C. Fault Types

LLTFI supports the injection of bitflips, which flips bits from 0 to 1 or vice versa, in the destination register of instructions. These bitflips can be single (i.e., one bitflip on a single instruction in a program) or multiple (i.e., multiple bitflips across multiple instructions or multiple bitflips in a single instruction). Users can also specify the bit position in which fault injection is performed. All of these options can be specified in a YAML (yet Another Markup Language) file by the user, without requiring any changes to the ML model.

```
compileOption:
  instSelMethod:
    - customInstselector:
        include:
          - CustomTensorOperator    # Select custom LLVM pass for FI
        options:                    # FI in 3rd Relu layer
          - -layerNo=3              # Number of layer for FI
          - -layerName=Relu         # Name of Tensor Operator for FI

  tracingPropagation: True          # Enable instruction tracing?
  tracingPropagationOption:
    maxTrace: 250                   # Max number of instructions to trace
    mlTrace: False                  # Enable tracing for ML programs?
runOption:
  - run:
    numOfRuns: 25                   # Number of experiments
    fi_type: bitflip                # Type of fault to inject
    fi_max_multiple: 1              # Number of faults to inject
```

Fig. 2: Example of the YAML file used by LLTFI to configure the FI experiments.

Fig. 2 shows an example of the YAML file. There are two sections in the YAML file: (1) compileOptions, and (2) runtime options. These mirror those used in LLFI [12]. Compiler options are those that determine which instructions should be selected for FI, as well as the layers of the ML model to inject into. Additionaly, they also specify whether to trace fault propagation, and if so, to what depth (i.e., number of instructions to trace). Runtime options, on the other hand, determine the FI experimental parameters, such as the number of experiments to be run, the type of fault to be injected, and the maximum number of bit-flips (for multi-bit faults) to inject. Due to space constraints, we do not show the complete list of options supported by the YAML file.

D. Tracing Fault propagation

LLTFI provides fault propagation tracing at two different levels of granularity: at the tensor operator level and the instruction level. Tensor operator level fault propagation tracing enables LLTFI to identify the effect of the injected fault on the output of the tensor operators. It can be used to determine how the fault originates, propagates, or gets suppressed by the tensor operator. On the other hand, instruction-level tracing can be used to precisely track the propagation of the injected fault through the assembly instructions at runtime (see Section IV).

IV. Example of running LLTFI on a ML Program

In this section, we illustrate how LLTFI works on a ML model for low-level fault injection (FI), and how it helps users visually understand the effects of FI. Suppose we have a ML model written in Python, using the TensorFlow and Keras [26] APIs, as shown in Listing 1[3]. This model is trained on the MNIST [27] dataset and aims to classify handwritten images of digits into a single numerical value between 0 and

[3]Though the example uses the TensorFlow framework, LLTFI is agnostic of the ML framework as it uses ONNX.

9. We compile this model, train it and export its weights and architecture information to a saved model format.

LLTFI's compile script automatically converts the saved model into LLVM IR, as depicted in Fig. 1. In this example, however, we break down the internal steps. The saved model is first converted into ONNX using the `tf2onnx` tool, resulting in model.onnx. Then, model.onnx is lowered into a MLIR file, model.mlir, using the ONNX-MLIR tool. The MLIR is very similar to LLVM IR [13]. Finally, model.mlir is converted into LLVM IR, *model.ll*, using the `mlir-translate` tool.

Listing 1: model.py - TensorFlow model

```
model = models.Sequential()
model.add(
  layers.Conv2D(32, (5, 5), activation="relu", ↩
      input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (5, 5), activation="↩
    relu"))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(10, activation="softmax"))
```

Listing 2: controller.c - to invoke TensorFlow model

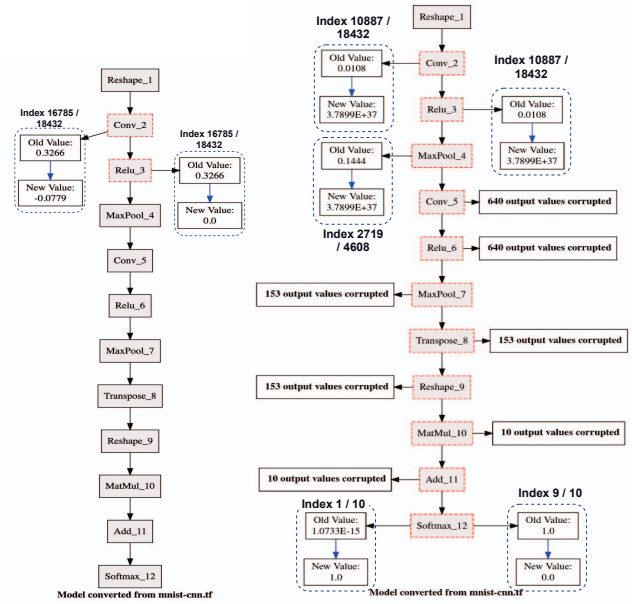
```
OMTensorList *run_main_graph(OMTensorList *);

int main(...) {
  OMTensorList *input = read_input(image_file);
  OMTensorList *outputList = run_main_graph(input);
}
```

Despite obtaining the LLVM IR code, model.ll cannot be executed directly. The model is contained within a function called *main_graph()*, which is invokable by the *run_main_graph()* function. A controller program reads the image file into a tensor, and invokes the model with the input tensor, as shown in Listing 2. This controller program is provided by LLTFI, and is independent of the ML model and dataset. Therefore, users do not need to write their own.

We compile the controller program using LLVM's clang to obtain controller.ll, which we then link with model.ll, to obtain a single unified LLVM IR file, *program.ll*.

Finally, we apply LLTFI to perform FI on *program.ll*. We first instrument it to add fault injection calls and profile it (i.e., run the instrumented code without any faults). In this example, we pass in a handwritten image of the digit 8 to the controller program, as shown in Fig. 1. The output of the model is an array of length 10, representing the digit-wise softmax output by the ML model. In the figure, we show the classes inferred with the highest probability. During the profiling run, LLTFI generates a golden output of $[0, ..., 0, 0, 0.999989, 0]$. This means that the model has classified the test image as the digit 8 with a probability of 0.999989. However, when the fault is injected, LLTFI generates an erroneous output of $[1, ..., 0, 0, 0, 0]$. Instead of correctly classifying it as 8, the model misclassifies the image as 0, thus resulting in a SDC.

LLTFI also allows users to trace the propagation of the injected fault among different layers of the ML application in the form of a data flow graph (DFG). We use Graphviz [28] to generate the visual DFGs. In Fig. 3b, we show the tensor operator-level error propagation DFG generated by LLTFI for this example. The DFG nodes highlighted with gray represent the layers of the ML model, and the other nodes originating from the layer nodes show the change in the layer output due to the injected fault. We observe that LLTFI injected a fault in the *Conv_2* layer thereby modifying the output value at index 10887 from 0.0108 to $3.7899E + 37$. This fault propagated through *Relu_3* and *MaxPool_4* layers, and at *Conv_5* layer, the fault gets propagated manifold thereby corrupting 640 different output values, and results in an SDC.



(a) Fault got suppressed by ReLu.

(b) Fault resulted in SDC.

Fig. 3: LLTFI on CNN-MNIST. First snippet shows the case where the fault got suppressed by the ReLu layer. Second snippet shows fault propagation resulting into a misclassification.

Fig. 3a shows another possible scenario where the injected fault corrupts a single output value of the *Conv_2* layer, but the fault got masked by the *Relu_3* layer. This fault results in a benign output. LLTFI's graph visualization feature offers an intuitive way to understand how low-level errors originate and propagate through different layers of the ML model.

V. Evaluation Methodology

In this section, we describe our experimental setup, followed by the research questions we asked in this work.

A. Experimental Setup

ML applications. We choose five ML programs, all of which are classifier models. These programs are of varying sizes and are commonly used by other FI studies [1]. We also included Nvidia's Dave2 [29], a Convolutional Neural Network (CNN) used in autonomous vehicles, which is a regression model. Table I lists the benchmarks and datasets.

TABLE I: ML applications used for LLTFI's evaluation.

Network	Dataset(s)	Number of parameters
CNN	MNIST, F-MNIST	44426
LeNet [31]	MNIST, F-MNIST	44426
VGG16 [32]	MNIST	14913226
AlexNet [33]	CIFAR10	1642282
SqueezeNet [34]	CIFAR10	122986
Dave2 CNN [29]	Driving [30]	252219

Datasets. We used three publicly available datasets for the classifier models: MNIST (hand-digit recognition), Fashion-MNIST (F-MNIST; fashion products classification), and CIFAR-10 (a large object recognition dataset). We also used a real-world driving dataset with labelled steering angles [30], for evaluating the regression model, Dave2.

Metrics. We measure the efficacy and performance overhead of LLTFI. We use SDC rate for measuring the efficacy based on the application type. For the classifier applications, the SDC rate refers to the fraction of injected faults that result in object misclassification. For the regression model, Dave-2, (i.e., predicting steering wheel angles), we use four different thresholds: 15, 30, 60, and 90, and classify an output as an SDC if the difference between the corrupted and the original output is greater than the threshold, as done by prior work [1].

For the performance overhead of LLTFI, we measured the profiling time and FI time of using LLTFI on our benchmarks (20 samples for each). Profiling time is a one-time cost to analyze the program (e.g., obtain instruction counts) at runtime. In contrast, FI time is a recurring cost and is measured as the time taken to run the fault injected program compared to the baseline program execution time.

Optimizations and Transformations. The default graph transformations by ONNX are semantics-preserving[4], and while converting the ML models to ONNX, we did not use any optimizations like constant folding. Moreover, while converting ONNX models to LLVM IR using ONNX-MLIR, we used only O0 optimization, thereby ensuring that the FI results from LLTFI are comparable to those from FI performed on the original ML program.

Setup. We ran all our experiments on an Intel Xeon E-2224 quad-core CPU with 900MHz clock speed. To keep our results comparable with the prior work [1, 3], we decided not to use GPUs for calculating the performance overheads. Moreover, for our ML benchmarks, we used FLOAT32 as a data type.

B. Research Questions

We asked four research questions (RQs) to compare high-level (using TensorFI) and low-level ML FI (using LLTFI).

- **RQ0** Are the FI results by LLTFI consistent among ML models written in different ML frameworks?
- **RQ1** How does the overall SDC rate of the ML model with LLTFI compare with that of TensorFI ?
- **RQ2** Does a single bit-flip fault injected in a specific layer of the ML model affect the output of that layer differently in both LLTFI and TensorFI?
- **RQ3** How does the performance overhead of LLTFI compare with that of TensorFI?

[4]https://onnxruntime.ai/docs/performance/graph-optimizations.html#basic-graph-optimizations

C. Fault Injections

Although LLTFI can be used to inject both single and multi-bit flip faults, in our FI experiments, we injected only single bit-flip faults as this is the de-facto fault model used in soft-error studies [17, 35, 36]. Further, Sangchoolie et al. have shown that single-bit faults are often sufficient for measuring the SDC rate of programs [37]. We injected faults only in the operands and results of floating-point arithmetic, logic instructions like `FADD`, and `FCMP`, to prevent application crashes (using the LLVM pass described in Section III).

To ensure an apples-to-apples comparison between LLTFI and TensorFI, we chose the same type of faults for each comparison point. However, in LLTFI, the faults were injected into the instruction operands and results, within an operator of the ML model. Whereas in TensorFI, the faults were injected directly into the output of a randomly chosen operator. This is because TensorFI (like most ML FI tools) does not provide visibility into the computations performed within an operator.

Note that only one fault was injected in each FI trial to avoid interference. Further, we report only those trials in which the faults were activated (i.e., read by the system). These assumptions are common in most FI studies.

VI. Evaluation

We organize the results based on the RQs.

A. *RQ0: FI consistency between ML frameworks*

For this RQ, we were interested in checking whether the FI by LLTFI is consistent among different ML frameworks. We chose two of the widely used ML frameworks - PyTorch and TensorFlow. We then compared the SDC rates reported by LLTFI for single-bit flip faults injected into the same ML models written using these two ML frameworks. We perform this experiment 1250 times for each framework, and program. We report the average SDC rates for each program, and the error bars at the 95% confidence intervals.

Fig. 4 shows the SDC rates for each of the benchmarks written using the TensorFlow and PyTorch frameworks (blue and green bars in the graph). We observed that the SDC rates are similar across all benchmarks (i.e., are within the error bars). *Therefore, LLTFI is consistent across the TensorFlow and PyTorch frameworks.*

Upon further inspection of our benchmarks, we found that the ONNX graphs exported by PyTorch and TensorFlow are very similar: most of the tensor operators are the same except `AveragePooling` and General Matrix Multiplier (`GEMM`). Unlike TensorFlow, PyTorch's ONNX exporter emits an extra padding operator with `AveragePooling`, and also prefers to use `GEMM` operator instead of `MatMul` operator for fully-connected layers. Nevertheless, these differences are minor

and do not change the overall program's semantics. Therefore, LLTFI's results are consistent between these ML frameworks. For the rest of the RQs, we will use only the TensorFlow versions of the benchmarks as the differences between the PyTorch and TensorFlow versions are negligible.

B. *RQ1: Comparison between high-level and low-level ML FI*

To answer this RQ, we injected single bit-flip faults using LLTFI and TensorFI in randomly-selected layers of the benchmark programs, and recorded the SDC rates. For each benchmark, we ran this experiment 1250 times, and reported the average SDC rates as well as error bars at the 95% confidence intervals.

Fig. 4 shows the SDC rates reported by LLTFI and TensorFI across all the benchmarks. As can be seen from the figure, across the benchmarks, the SDC rates reported by TensorFI were much higher than those reported by LLTFI. The differences ranged from $1.1X$ (LeNet-MNIST) to $5.5X$ (SqueezeNet-CIFAR10), with an average of $3.4X$. Thus, TensorFI over-reports the SDC rate compared to LLTFI. We examine the potential reasons for the same in the next RQ.

C. *RQ2: Differences between the tools for a bit-flip fault in a specific layer of the ML model affecting the layer output*

TABLE II: Error Propagation of single bit-flip faults injected by LLTFI across models.

Benchmark Name	% of faults that affected layer's output	Corrupted element differs from original by one bit flip
CNN-MNIST	37.9%	19.2%
CNN-FMNIST	37.6%	23.7%
LeNet-MNIST	38.1%	29.5%
LeNet-FMNIST	39.7%	30.2%
VGG16-MNIST	37.0%	24.4%
AlexNet-CIFAR10	43.4%	27.2%
SqueezeNet-CIFAR10	43.0%	30.0%
Dave2-Driving	38.1%	26.6%
Average:	**39.4%**	**26.4%**

For this RQ, we were interested in the following questions: (1) what percentage of bit-flip faults injected by LLTFI propagate to the output of the target layer? (2) can a single bit-flip fault corrupt multiple output elements of the target layer?, and (3) how often does the corrupted output element(s) value differ from the correct output by just a single bit-flip?.

These questions will help us answer why LLTFI leads to much lower SDCs than TensorFI, as we found in RQ1. For example, if it turns out most faults injected in a layer by LLTFI get masked before reaching the output of a layer, that could explain why LLTFI has a lower SDC rate than TensorFI, which always injects a fault into the output layer. Conversely, if a single bit flip fault injected by LLTFI reaches the output of the layer but ends up corrupting multiple elements of the output, then it could have a more significant effect on the SDC rate than TensorFI, which typically corrupts a single element of the layer output.

TABLE III: Profiling and fault injection (FI) overheads of LLTFI for ML applications.

Benchmark Name	Baseline Time (s)	Profiling Overhead	FI Overhead	TensorFI Baseline (s)	TensorFI FI Overhead
CNN-MNIST	0.088	51%	48%	1.51	85%
LeNet-MNIST	0.045	98%	113%	1.53	77.40%
CNN-FMNIST	0.056	69%	86%	1.56	73%
LeNet-FMNIST	0.045	104%	117%	1.50	76%
VGG16-MNIST	0.982	4%	19%	2.15	138%
AlexNet-CIFAR10	0.331	10%	39%	1.71	81%
SqueezeNet-CIFAR10	0.081	82%	81%	2.40	80%
Dave2-Driving	0.363	12%	38%	1.52	82%
	Average Overheads:	**54%**	**68%**		**86.6%**

To answer this RQ, we injected a single bit-flip fault using LLTFI in a randomly-selected layer of our benchmarks, and studied the error propagation using LLTFI's fault tracing capabilities. For this experiment, we ran each FI campaign 1000 times for each benchmark and calculated the average percentages of occurrences of each of the above outcomes.

The results are shown in Table II. We observed that, on average, only 39.4% of the faults injected by LLTFI propagate to the output(s) of the target layer. The remaining faults are either masked primarily due to multiplication by zero (for convolution and matrix multiplication layers), or are pruned due to the activation function or max-pooling layers. Moreover, in *all of our experiments*, we observed that a single bit-flip fault corrupted at most a single output element of the target layer (not shown in the table). In other words, *there was no case in which a single bit-flip fault propagated to multiple output elements of the ML model layer.* These observations are the reason why the SDC rate of LLTFI is significantly lower than that of TensorFI for the same fault types.

Finally, we find that though we injected only a single bit-flip fault in each layer, it resulted in a single bit flip in the output of the layer in only 26.4% of the cases on average. In the remaining 73.6% cases, the corrupted output element of the layer differs from its original value by multiple bit-flips. TensorFI injects single bit flips into the output elements of the layer. However, low-level faults often lead to multiple bit-flips at the output layer of the ML model. In Section VIII, we further discuss these results along with their implications.

D. *RQ3: LLTFI's performance overheads*

We measure the performance overhead due to profiling and FI by LLTFI in Table III. Recall that profiling is a one-time cost, whereas FI is a recurring overhead. We find that the overhead due to profiling is 58% on average (ranges from 12% to 104%), while the FI overhead is 68% on average (ranges from 19% to 117%) compared to the baseline model.

We also compare the overhead of LLTFI with TensorFI for each benchmark in Table III. Note that TensorFI does not have a separate profiling phase, so the overheads for TensorFI represent the FI overheads. As can be seen, the average overhead of TensorFI is 86.6%, compared to 68% for LLTFI. Note that the baseline execution times of TensorFI are also much higher than those of LLTFI. However, the FI overhead is calculated as a percentage of the baseline time.

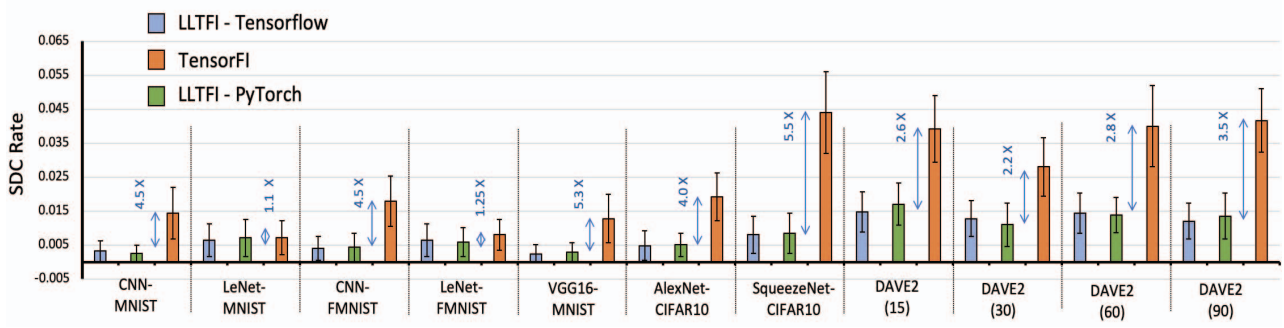


Fig. 4: Average SDC Rate for single bit-flip faults injected by TensorFI and LLTFI across benchmarks. For LLTFI, we consider both the TensorFlow and PyTorch versions of the benchmarks. The error bars show the 95% confidence intervals.

Thus, LLTFI is $1.27x$ faster than TensorFI, on average. This is because LLTFI works at the LLVM IR level for instrumenting the code for FI, whereas TensorFI instruments the Python code. The LLVM IR code gets compiled to native assembly code, as opposed to Python code, which is interpreted. Further, the instrumentation added by LLTFI can be optimized by the LLVM compiler, while the instrumentation added by TensorFI can inhibit compiler optimizations.

VII. Case Study: Extending LLTFI for Error Detection and Correction

Selective Instruction Duplication (SID) is a technique for selectively duplicating certain instructions in the program to detect soft errors [38]. SID has been studied in the context of general-purpose programs [39], GPUs [36], Embedded systems [40], super-scalar processors [38], etc. In this study, we use LLTFI to evaluate the effectiveness of SID techniques for ML applications in a framework-agnostic fashion. Further, we go beyond error detection and also actively correct the error with SID. We extended LLTFI to perform SID at the LLVM IR level, and add correction logic to the application.

To determine which instructions to duplicate, we used LLTFI to find the sensitivity (SDC rate) and the vulnerability (fraction of program run time spent in that instruction) of each arithmetic instruction within the ML program. We observed that arithmetic instructions within the convolution operators are the most vulnerable and sensitive ones. Similar findings have been reported by prior work [41, 42], and therefore, for this case study, we selected the arithmetic instructions within the convolution operators to duplicate with SID.

We then apply the three SID heuristics discussed in the next section to duplicate the instructions.

A. Duplication Heuristics

We use three heuristics for duplicating the instructions.

1) Arithmetic SID (AID): Since we assume that transient hardware faults do not modify the control flow structure of the ML applications (Section II-B), it is sufficient to duplicate only the arithmetic instructions such as `FMUL`, `FSUB`, `FDIV`, and `FADD`, which are primarily involved in the computation of different DNN operators such as convolution and GEMM (General Matrix Multiplication).

As shown in Figure 5a, our extension of LLTFI duplicates arithmetic instructions like `FMUL` and `FADD`, and then inserts the logic to compare the outputs of the original (`Val #1`) and the duplicated instructions (`Val #2`). Upon detecting a mismatch, the error correction heuristics described in Section VII-B are deployed.

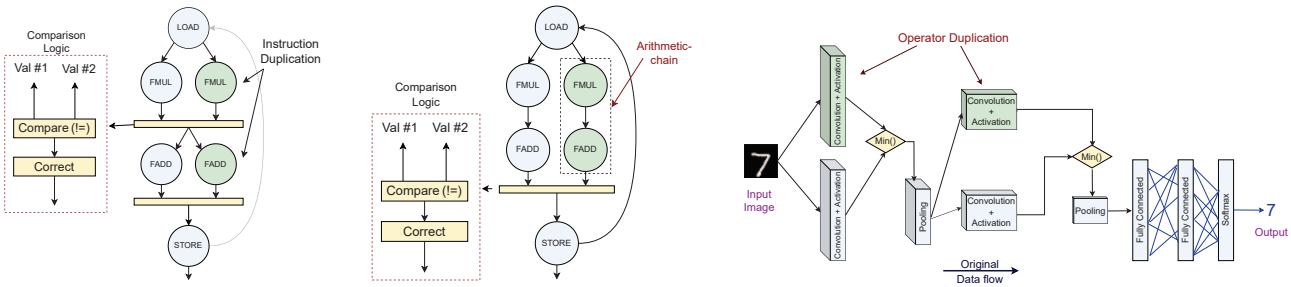
2) Arithmetic Chain Duplication (ACD): ACD is an optimization of AID that aims to reduce the number of comparison operations in the code. Referring to Figure 5b, ACD works by (1) statically identifying and duplicating the chain of arithmetic instructions, and (2) adding the error detection and correction right after the chain. An arithmetic instruction chain consists of a sequence of arithmetic instructions within a basic block, such that the output of each instruction (except the last one) only affects the input of one or more arithmetic instructions within that chain.

3) Operator Duplication (OD): Operator duplication, works by fully duplicating the neural network's layer(s). As shown in Figure 5c, operators such as convolution and matrix multiplication are fully duplicated, followed by the comparison logic to compare the output of the original and the duplicated operators. We implemented operator duplication at the ONNX level instead of the LLVM IR level.

B. Error correction heuristics

Upon detecting a mismatch, we use two heuristics for error correction: *bitwise-and* and *return-min*. The former heuristic returns the *bitwise-and* of the two values while the *return-min* heuristic returns the minimum of the two values. These values are used in the subsequent computations. As noted by Li et al. [35], soft errors affecting the most significant bits have the highest probability of causing SDCs. Therefore, both the correction heuristics aim to prevent large, positive values from propagating through the CNN, thus preventing unnaturally large fluctuations in the neuron's output.

Note that for OD, we only use *return-min* correction heuristic as *bitwise-and* heuristic is not a standard operation in the



(a) Arithmetic SID (b) Arithmetic Chain Duplication (c) Full Operator Duplication

Fig. 5: Three different SID heuristics in our work. The nodes highlighted in blue represent the original data flow of the program, while those in green depict the duplicated nodes, and the ones in yellow belong to the comparison logic.

ONNX framework[5] and, thus, is not supported by the ONNX-MLIR tool for conversion to the LLVM IR level.

C. Evaluation

Methodology. To evaluate the effectiveness of the SID extension of LLTFI, we used the six benchmarks from Table I. Furthermore, we used the same evaluation metrics and setup described in Section V. We performed each FI experiment 1000 times with LLTFI, and report the median values and error bars with 95% confidence intervals as before.

Results. The objective of our evaluation is to *evaluate the feasibility of SID for ML applications.* Towards this goal, we evaluate the trade-offs of SID concerning its reliability benefits (SDC Rate), and runtime execution overheads.

Figure 6 shows the SDC rate reduction percentage from the original, unprotected program of the ACD SID technique with *bitwise-and* and *return-min* correction heuristics. We observe similar SDC rates for AID SID, and hence, omitted it from Figure 6. The best-case reduction in the SDC rate for the benchmarks varies from 85.7% (for LeNet-MNIST) to 100% (for AlexNet and Dave2). On average, we observed a 93.9% best case reduction in the SDC rate across all our benchmarks. We also observed the lowest SDC rate for ACD SID with *return-min* correction heuristic, except for AlexNet-Cifar10. Moreover, the SDC rate reduction is comparable for OD and ACD with *return-min* correction heuristic.

To understand these results, we manually inspected the faults that caused the ML program to misclassify, despite our error correction heuristics. We made two observations.

1. For *return-min* correction heuristic, we observed that the majority of the escaped faults are NaN (Not a Number) values. The IEEE-754 floating-point standard defines a NaN as a number with all ones in the exponent and a non-zero mantissa. Arithmetic instructions (except bitwise manipulations) and logic instructions operating on a NaN value will always result in a NaN value, and thus, these faults escaped our *return-min* correction heuristic.

2. For *bitwise-and* heuristic, interestingly, the NaN values got "corrected" by the bitwise-and operation because the bitwise-and of the original and the corrupted value prevents the situation with all exponent bits equal to one. However, unlike the *return-min* heuristic which consists of only logical assembly instructions, *bitwise-and* heuristic also consists of arithmetic instruction (for bitwise `AND` operation), which itself is susceptible to soft errors. Thus, the faults that escaped *bitwise-and* correction heuristics are the ones that corrupted the `AND` operation itself.

Figure 7 shows the runtime overhead of AID, ACD SID techniques with *bitwise-and* and *return-min* correction heuristics. The best-case runtime overhead varies from 4.43% (for AlexNet) to 40% (for Dave2). Moreover, the least runtime overhead was observed for ACD with *bitwise-and* heuristic. This is because ACD reduces the number of comparison logic, and *bitwise-and* heuristic can be implemented with just a single assembly instruction. Moreover, we also noticed a significant reduction in runtime overheads with SID, as opposed to full operator duplication (proposed by Libano et al. [42]). Overall, these results suggest that for benchmarks like AlexNet and Vgg16, SID can significantly improve the error resilience (reduction in SDC rate by 90% and 100%) with only modest runtime overheads (7.8% and 4.4%).

For the Dave2 benchmark, we observed that the second convolution layer is very computationally expensive, and takes about 52% of the total program runtime. Therefore, SID in this convolution layer resulted in an execution overhead of 40%.

Overall, SID achieves high reduction in SDC rate with a modest performance overhead for most of our benchmarks.

VIII. Implications

a) Implication 1: Overreporting of SDCs: Our experiments for RQ1 show that TensorFI reports a significantly (1.1x - 5.5x) higher SDC rate than LLTFI for the same types of hardware faults. *This is due to TensorFI's assumption that every bit flip fault corrupts a single layer output.* However, our experiments with LLTFI contradict this assumption, as we find that only a fraction of faults (about two out of five) ends up corrupting the layer output. These results potentially apply to other ML FI tools like PyTorchFI that inject faults in the layer outputs. Therefore, when using ML FI tools, one needs to

[5]Following is the list of tensor operators supported by ONNX: https://github.com/onnx/onnx/blob/main/docs/Operators.md

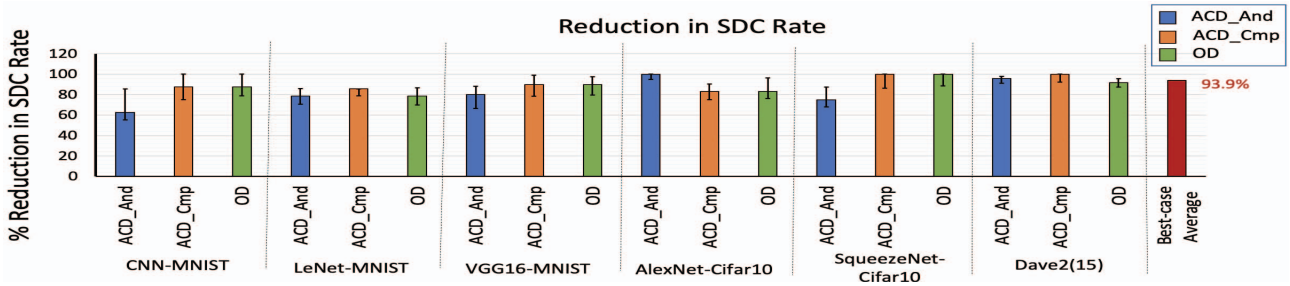


Fig. 6: Percentage reduction in the SDC Rates for LLTFI with various SID techniques and correction heuristics compared to the unprotected program. *ACD_Cmp* and *ACD_And* are the shorthands used for ACD with *return-min* and *bitwise-and* heuristics respectively. *OD* stands for the operator duplication technique. SDC rates for AID and ACD were similar, hence, not shown separately. The bar in red shows the best-case average SDC reduction across all our benchmarks. Higher values are better.

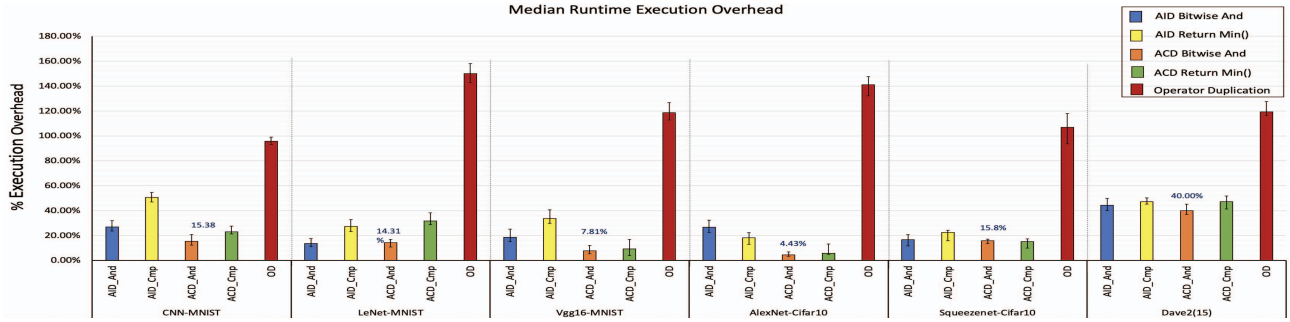


Fig. 7: Runtime Execution Overheads for LLTFI with various SID techniques and correction heuristics. Shorthand notations similar to Figure 6. The percentages are relative to the original, unprotected program. Lower values are better.

scale down the SDC rate estimates to obtain realistic estimates. However, the amount of scaling is application specific.

b) Implication 2: Faults in layer outputs: We also found that the majority of low-level faults result in multiple bit-flips in layer outputs. ML FI tools such as TensorFI primarily inject single bit flip faults into layers' outputs. However, we found that low level faults corrupted at most one value in a layer's output (RQ2), so the choices made by ML FI tools to corrupt a single value in the output layer are well justified.

c) Implication 3: Evaluation of low-level resilience enhancing techniques (like SID) for ML applications: To the best of our knowledge, we are the first ones to evaluate the efficacy of SID for ML programs (Section VII). Although not exhaustive, our preliminary results are promising: for a few benchmarks like AlexNet and Vgg16, SID (in the convolution operator) significantly reduces the SDC rates (90%-100%) with modest runtime overheads (7.8% - 4.4%). LLTFI thus enables researchers to evaluate different SID heuristics and identify the instructions to duplicate. LLTFI is able to evaluate these and other low-level resilience improvement techniques [43], due to its visibility into instructions and registers, which is missing in most ML FI tools.

d) Overall Implications: Prior research on improving the error resilience of ML applications [16] [17], [44] [45] [46] use high-level ML FI tools for evaluating their techniques. Our results suggest that there is a need to revisit the evaluation of these techniques. One of the main advantages of high-level ML FI tools is that they allow abstraction of the faults to the ML framework's level, and allow programmers to study propagation of these errors at that level. However, LLTFI offers similar advantages as high-level ML FI tools, without compromising on the representativeness of the injected faults.

IX. Related Work

The closest ML FI tools to LLTFI are PyTorchFI [3], TensorFI [1], and MindDI [2] developed for the PyTorch, TensorFlow, and Mindspore frameworks respectively. These tools are specific to their respective ML frameworks, whereas LLTFI works with any ML framework that can be compiled with ONNX and MLIR. Further, all of these tools can only inject into the ML model parameters and the outputs of operators at the framework level, rather than into specific registers and instructions like LLTFI can. Finally, none of the tools can inject into specific instruction types, as these are typically not visible at the ML framework level.

Mahmoud et al. [47] recently proposed a PyTorch-based FI tool, GoldenEye, to inject bit-flip faults in the hardware-like representation of data types for PyTorch applications. Unlike us, they evaluate performance-resilience tradeoffs of using data types like Signed Fixed-point (FxP), Block floating-point

(BPF), and Adaptive floating-point (AFP) in ML applications. However, they do not support injecting faults into individual instructions in a framework-agnostic manner like LLTFI.

Reagen et al. presented Ares [20], an application-level fault injector that optimizes fault injection speed by using tensor operations on the GPU. In contrast to LLTFI, Ares changes the Keras inference computation, thereby requiring that the ML application be rewritten to use the new interface.

Li et al. [35] characterizes the impact of soft hardware errors on deep neural networks (DNN) by performing systematic, micro architectural-level FI using a DNN simulator. While micro architectural-level FI is more precise, it is hardware-dependent and is time- and resource-intensive for most applications.

X. Conclusions

This paper presents LLTFI, a low-level fault injection tool for ML models that is ML-framework agnostic. LLTFI's tensor operator level fault propagation tracing offers an intuitive understanding of low-level error propagation through different ML model layers. LLTFI is configurable through a YAML file, and does not require any code modifications or annotations.

We evaluate LLTFI with six popular ML programs, and compare it to TensorFI, a high-level FI tool for ML programs. We find that TensorFI overreports the Silent Data Corruption (SDC) rate of these programs for single bit-flip faults by 3.5X on average compared to LLTFI. Further, there are significant differences in fault propagation between the two tools. Finally, LLTFI is 27% faster than TensorFI on average. We also demonstrate the utility of LLTFI by extending it to perform selective instruction duplication for ML applications.

Acknowledgements

This work was funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Huawei Corporation. We thank the ISSRE'22 reviewers for their insightful comments.

References

[1] G. Li, K. Pattabiraman, and N. DeBardeleben, "TensorFI: A Configurable Fault Injector for TensorFlow Applications," in *Proc. of ISSREW'18*, 2018.

[2] Y. Zheng, Z. Feng, Z. Hu, and K. Pei, "Mindfi: A fault injection tool for reliability assessment of mindspore applications," in *Workshop on Dependability Modeling and Design*, 2021.

[3] A. Mahmoud *et al.*, "PyTorchFI: A Runtime Perturbation Tool for DNNs," in *Proc. of DSN-W'20*, 2020.

[4] H. Ziade, R. A. Ayoubi, and R. Velazco, "A Survey on Fault Injection Techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[5] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[6] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, 2019.

[7] N. Narayanan and K. Pattabiraman, "TF-DM: Tool for Studying ML Model Resilience to Data Faults," in *Proc. of DeepTest'21*, 2021.

[8] "Baidu Apollo team (2017), Apollo: Open Source Autonomous Driving, howpublished = https://github.com/apolloauto/apollo, note = Accessed: 2022-05-10."

[9] "comma ai openpilot: an open source driver assistance system, howpublished = https://github.com/commaai/openpilot, note = Accessed: 2022."

[10] Z. Peng, J. Yang, T.-H. P. Chen, and L. Ma, "A First Look at the Integration of Machine Learning Models in Complex Autonomous Driving Systems: A Case Study on Apollo," in *Proc. of ESEC/FSE '20*.

[11] A. Chan, U. K. Agarwal, and K. Pattabiraman, "(wip) lltfi: Low-level tensor fault injector," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2021, pp. 64–68.

[12] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults," in *Proc. of QRS '15*, 2015.

[13] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of CGO'04*.

[14] C. Lattner *et al.*, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *Proc. of CGO'21*, 2021.

[15] J. Bai, F. Lu, K. Zhang *et al.*, "ONNX: Open Neural Network Exchange," https://github.com/onnx/onnx, 2019.

[16] Z. Chen, G. Li, and K. Pattabiraman, "A Low-cost Fault Corrector for Deep Neural Networks through Range Restriction," in *Proc. of DSN'21*.

[17] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "BinFI: An Efficient Fault Injector for Safety-Critical Machine Learning Systems," in *Proc. of SC'19*, 2019.

[18] L. Matanaluza *et al.*, "Emulating the Effects of Radiation-Induced Soft-Errors for the Reliability Assessment of Neural Networks," *IEEE Transactions on Emerging Topics in Computing*, 2021.

[19] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *Proc. of DSN'12*, 2012.

[20] B. Reagen *et al.*, "Ares: A Framework for Quantifying the Resilience of Deep Neural Networks," in *Proc. of DAC '18*, 2018.

[21] T. Jin *et al.*, "Compiling ONNX Neural Network Models Using MLIR," 2020, arXiv:2008.08272.

[22] N. Rotem *et al.*, "Glow: Graph Lowering Compiler Techniques for Neural Networks," *CoRR*, 2018.

[23] C. Leary and T. Wang, "XLA," 2017.

[24] T. Chen *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *Proc. of OSDI'18*, 2018.

[25] Ragan-Kelley *et al.*, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," 2013.

[26] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[27] Y. LeCun, C. Cortes, and C. Burges, "MNIST handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[28] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph – static and dynamic graph drawing tools," in *GRAPH DRAWING SOFTWARE*, 2003.

[29] M. Bojarski *et al.*, "End to End Learning for Self-Driving Cars," 2016, arXiv:1604.07316.

[30] S. Chen, "Labeled car driving dataset," https://github.com/SullyChen/driving-datasets, 2021.

[31] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, vol. 20, no. 5, p. 14, 2015.

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

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

[34] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," 2016, arXiv:1602.07360.

[35] G. Li *et al.*, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," in *Proc. of SC '17*.

[36] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing Software-Directed Instruction Replication for GPU Error Detection," in *Proc. of SC'18*, 2018.

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

[38] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[39] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-Directed Instruction Duplication for Soft Error Detection," in *Proc. of DATE'05*, 2005.

[40] N. Oh and E. J. McCluskey, "Error detection by selective procedure

call duplication for low energy consumption," *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 392–402, 2002.
[41] M. Beyer *et al.*, "Fault Injectors for TensorFlow: Evaluation of the Impact of Random Hardware Faults on Deep CNNs," in *Proc. of ESREL'20*, 2020.
[42] F. Libano *et al.*, "Selective Hardening for Neural Networks in FPGAs," *IEEE Transactions on Nuclear Science*, vol. 66, no. 1, pp. 216–222, 2018.
[43] M. Didehban and A. Shrivastava, "nzdc: A compiler technique for near zero silent data corruption," in *Proc. of DAC '16*. IEEE, 2016.
[44] L.-H. Hoang, M. A. Hanif, and M. Shafique, "Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation," in *Proc. of DATE'20*, 2020.
[45] E. Ozen and A. Orailoglu, "Sanity-Check: Boosting the Reliability of Safety-Critical Deep Neural Network Applications," in *Proc. of ATS'19*.
[46] A. Mahmoud *et al.*, "Optimizing Selective Protection for CNN Resilience," in *Proc. of ISSRE'21*, 2021.
[47] A. Mahmoud, T. Tambe, T. Aloui, D. Brooks, and G.-Y. Wei, "GoldenEye: A Platform for Evaluating Emerging Numerical Data Formats in DNN Accelerators," in *Proc. of DSN'22*, Forthcoming.