TensorFledge: A Fault Injector for the EdgeML TensorFlow Library

EECE 571K: Security & Reliability in IoT Project Final Report

Niranjhana Narayanan
University of British Columbia
Vancouver, Canada
nniranjhana@ece.ubc.ca

***Abstract*—The Internet of Things has seen the adoption of edge computing and Machine Learning in diverse domains ranging from home and industrial automation to healthcare. Many of these systems are safety-critical, requiring safety guarantees on their outputs. Thus it has become necessary to evaluate the error resilience of the ML algorithms that are used in these applications. Faults in these systems can be both due to software or hardware, and fault injection is a popular technique to determine the reliability of such programs.**

In this work, we build a fault injector called TensorFledge for the EdgeML TensorFlow library. EdgeML includes recently developed ML algorithms specialized for edge computing such as ProtoNN, Bonsai, FastRNN, EMI-RNN, etc. Focusing on both intentional and unintentional hardware faults, we compare and contrast the fault tolerance for these 8 models. The results give us insights into which of the existing models are more error resilient and thus provide hints for fault tolerant design in developing new EdgeML models. For example, we find that having an additional set of RNN cell parameters improves the accuracy up to 55.11% in the presence of certain faults or having a gated unit improves performance even under high model parameter sparsity.

***Index Terms*—machine learning, edge computing, IoT, error resilience, hardware fault tolerance**

I. Introduction

The Internet of Things (IoT) has a growing number of devices deployed in healthcare, self-driving cars, aircraft flight control, industrial automation and smart homes. A dominant paradigm in these systems is the dumb IoT device that is restricted to sensing data and transceiving information between the cloud and its environment. But with the recent increased growth and adoption of Machine Learning (ML) and edge computing in IoT, there has been active alternate research in designing algorithms to run locally on even tiny, resource-constrained IoT devices [1] [2] without offloading classification, regression, ranking and other common ML/IoT tasks to the cloud.

Many of these IoT systems are increasingly employed in safety-critical applications and hence require fast and accurate guarantees on how their outputs behave for any given situation. Thus it is important to establish the reliability of these models in the presence of faults and failures. This could mean either software or hardware faults. A type of transient hardware faults or soft errors are predicted to increase with progressive technology scaling and lower operating voltages [3] [4]. This is especially relevant in low power micro-controllers used in IoT. Therefore the focus in our work is on hardware faults.

Hardware faults can be intentional or unintentional:

(i) Intentional hardware faults can occur via a myriad of techniques/attacks such as pin-level fault injection, radiation-based fault injection, electromagnetic interferences, power supply disturbances, etc. It is also possible to cause hardware faults through software (e.g., rowhammer attacks [5]).

(ii) Unintentional hardware faults can be soft errors (i.e., transient hardware faults) which occur due to cosmic ray strikes, causing chip-level errors (such as bit flips in memory cells) or system-level errors (such as noise processed as bad data), subsequently causing errors in programs which could propagate and cause faulty outputs.

Thus it becomes necessary to evaluate the resilience of the ML applications in the presence of such hardware faults before they are deployed on edge IoT devices.

Fault injection (FI) is a commonly used technique to achieve this. There have been numerous tools capable of emulating hardware faults through software [6] [7]. But there has been very little work in FI applied to ML [8], and no work in FI applied to ML in edge devices. This research gap could be attributed to a couple of reasons. Firstly, ML applications are written in specialized frameworks such as TensorFlow, PyTorch, Caffe, Theano, etc. making conventional program analysis difficult. Secondly, the ML models specific for IoT edge devices have only been developed recently, and is still an active area of research. Microsoft's Edge ML library [9] contains an open source implementation of these models in the most popular ML frameworks such as TensorFlow and PyTorch.

In this project, we build a fault injector, TensorFledge for the EdgeML TensorFlow library. We model certain hardware faults and use our tool to evaluate the error resilience of these EdgeML models. The inspiration for TensorFledge is drawn from TensorFI [8], a fault injector for TensorFlow applications. Due to certain fundamental differences in the way EdgeML models are built, we were not able to extend TensorFI for our applications. We discuss these details in a later section. TensorFI and its TensorFledge variants are easy to use and flexible, allowing programmers to inject faults through an external interface without modifications to their application.

The contributions in this work are two-fold:

(i) A systematic approach to perform fault injection in ML models developed for edge IoT devices using frameworks such as TensorFlow, and

(ii) An evaluation of error resilience in 8 EdgeML models with popular datasets under different hardware faults and fault probabilities.

The results give us valuable insights into which of the ML models used for edge IoT are more error resilient. We follow with a detailed discussion and comparative analysis to help users determine which perform better in the presence of certain faults so they can choose the appropriate models for their safety-critical applications. Consequently, this evaluation of error resilience also provides us with hints for robust and fault tolerant design in future models.

TensorFledge is made publicly available on GitHub[1], along with documentation on installation, usage and the experimental results achieved in this paper.

II. Background and Related Work

A. Machine Learning and Internet of Things

Before delving into fault injection, we briefly summarize the background needed for this work and motivate the need to evaluate the upcoming ML models for IoT devices.

1) ML Applications: Machine learning mainly consists of algorithms and statistical models that computer programs use to perform a specific task without using explicit instructions, relying on patterns and inference instead. ML builds this inference into the models and model parameters using patterns in training and sample data. ML is used in a wide variety of modern day applications [10], ranging from email filtering to self driving vehicles.

An input to the ML model would contain specific features that help the model to make the prediction. Prediction tasks can be divided into classification and regression. The former is used to classify the input into categorical outputs (e.g., image classification). The latter is used to predict dependent variable values based on the input. ML models can be either supervised or unsupervised. In the supervised setting, the training samples are assigned with known labels (e.g., linear regression, neural network), while in an unsupervised setting there are no known labels for the training data (e.g., k-means clustering).

An ML model typically goes through two phases:

(i) training phase: model is trained to learn a task;

(ii) inference phase: model is used to make predictions.

The parameters of the ML model are learned from the training data in the training phase, and the trained model will be evaluated on the test data in the inference phase, which represents the unseen data and will not be used in the training phase. Typically, a separate validation set (usually created from a subset of training data) is used to serve as an unbiased approximation of the test data to evaluate the models (since test data cannot be used in the training procedure to evaluate the models).

[1] https://github.com/nniranjhana/TensorFledge

2) IoT devices: The Internet of Things is poised to revolutionize our world. Billions of micro-controllers and sensors have already been deployed for predictive maintenance, connected cars, precision agriculture, personalized fitness and wearables, smart housing, cities, healthcare, etc [11]. With the rise in edge computing, and thereby shifting the intelligence from the cloud to the devices, it becomes necessary to develop a library of efficient machine learning algorithms that can run on severely resource-constrained edge and endpoint IoT devices ranging from the Arduino to the Raspberry Pi.

In addition, since the energy required for executing an instruction might be much lower than the energy required to transmit a byte, making predictions locally would extend battery life significantly and also prevent damage due to excess heat dissipation from the communicating radio. This also enables a number of critical scenarios, beyond the pale of the traditional paradigm, where it is not desirable to send data to the cloud due to concerns about latency, connectivity, energy, privacy and security.

3) ML algorithms for Edge IoT devices: Keeping the above motivations in mind, Microsoft's EdgeML algorithms include collections of k-nearest neighbour and tree based algorithms, called ProtoNN and Bonsai respectively, for classification, regression, ranking and other common IoT tasks. These can be trained on the cloud, or on a laptop, but can then make predictions locally on the tiniest of microcontrollers without needing cloud connectivity. The algorithms are:

(i) ProtoNN: A novel, k-nearest neighbors (kNN) based general supervised learning algorithm [1].

(ii) Bonsai: A new non-linear tree classifier model for supervised learning tasks such as binary and multi-class classification, regression and ranking [2].

(iii) FastRNN & FastGRNN: Efficient RNN training and accurate prediction algorithms especially for severely resource constrained IoT devices [12].

(iv) EMI-RNN: Multi-instance learning method with early stopping for fast and efficient classification of sequential data on tiny devices [13].

(v) Robust PCA: A general-purpose anomaly detection algorithm [14].

In a later section, we will discuss certain key features in the models and how they differ from conventional ML models that make them adept at low power classification with good levels of accuracy. We will also discuss how these features fare with respect to fault tolerance.

B. Software Fault Injection

Software fault injection (SFI) is an acknowledged method for assessing the dependability of software systems [15]. It is the injection of faults through software-based techniques such as debuggers and instrumentation. SFI can be used for injecting either hardware or software faults. In this paper, we focus on emulating hardware faults for our software fault injection. There are various methods employed in SFI, depending on the use case. Let us see a quick classification and the justification for the methods we employ.

SFI techniques can be classified into compiler-based or runtime injections. In compiler-based techniques, the code is mutated to inject the fault prior to the program being run. In runtime fault injection techniques, the perturbation is done during the execution of the program with no changes to the code during its compilation. Runtime injection has the advantage that it can access the dynamic state of the program but is limited in that it does not have information about where the fault can occur in the program. Compile-time injection on the other hand does not have access to runtime information, but it can use knowledge of the program's code to seed the fault. It is also typically faster than runtime injections. Hybrid FI attempts to combine the advantages of compile-time and runtime injections by modifying the code at compile-time to insert hooks for runtime injections.

FI techniques can also be classified into interface-level injection and implementation-level techniques. In the former, faults are injected at the level of interfaces (e.g., function calls, Application Programmer Interface (APIs)), while in the latter, faults are injected into the implementation of the APIs or functions. The main advantage of interface-level fault injection is that it is independent of the implementation and hence more portable than implementation-level fault injection. On the other hand, interface-level injection is often more limited than implementation level injection as it cannot access the internal states of the SUTs. With that said, it is possible to emulate a wide variety of faults that propagate to the interface of the SUT, and hence interface-level injection is widely used in practice [6], [7]. In this paper, we primarily inject faults at the interface-level of APIs in TensorFlow. This gives us the portability advantage.

C. TensorFlow Framework

TensorFlow is an open-source framework for modeling large dataflow graphs and has been widely used for constructing ML algorithms. TensorFlow allows programmers to represent the program in the form of a TensorFlow graph. This distinguishes it from some other frameworks such as Keras, which only provide an abstract API for expressing the ML algorithms. TensorFlow is much more flexible and can be better optimized as it exposes the underlying graph to the developer. Thus, TensorFlow is considered a lower level framework. Many of the higher level frameworks such as Keras use TensorFlow as their back-end for implementation.

To use TensorFlow, programmers first use one of the in-built learning algorithms to construct the dataflow graph of their ML algorithm during the training phase. Various optimizations on the graph are also performed based on the input data set. Once the graph is built and optimized, it is not allowed to be modified. The graph is then used for the inference phase. Here the data is fed into the graph through the use of placeholder operators, and the outputs of the graph correspond to the outputs of the ML algorithm. In this phase, the graph is typically executed directly in the optimized form on the target platform using custom libraries.

TensorFlow also provides a convenient Python language interface for programmers to construct and manipulate the dataflow graphs. Though other languages are also supported, the dominant use of TensorFlow is through its Python interface. Note however that the majority of the ML operators and algorithms are implemented as C/C++ code, and have optimized versions for different platforms. The Python interface simply provides a wrapper around these C/C++ implementations.

D. Related Work

Prior research has found design/software bugs in ML programs through traditional software testing approaches. These include random testing, metamorphic testing, mutation testing or a combination of these [16] [17] [18]. However these do not examine the effects of hardware faults.

It has been shown possible to reverse engineer the model parameters such as weights and biases and/or even modify them through side channel attacks, using the timing information from power consumption or electromagnetic emanation [19]. SNIFF targets only one approach (deep-layer feature extractor in teacher-student transfer learning) and determines the last layer's weights and biases by attacking them and observing the differences between original and faulty outputs with multiple runs. But more importantly, evaluating the model's resilience in the presence of faulty parameters, is left unexplored in the author's work. The attack is performed by a bit flip on the sign bit of either the product of input and weight, or the bias value. In our work, we consider fault injection on the model parameters for bit flips and as well as random faults.

Frameworks have been developed to evaluate the resilience when there are static faults in memory through software fault injection, similar to our goal. Ares [20] finds that there are four fault points in a DNN: weights, activation, states and the data path. They evaluate on 6 DNN models but have not considered the fault points in ML models for edge devices. In our work, we define the fault model and fault points according to the ML algorithms used for edge IoT computing.

There has been work which examines the effects of hardware fault attacks through software in deep neural networks (DNN) [21]. However, they do not consider ML programs for edge computing or embedded IoT devices. Further, they do not have a way of systematically injecting faults and observing the error resilience under different fault occurrences.

In our work, we build a fault injector called TensorFledge to achieve exactly this, taking inspiration from the open source fault injector for TensorFlow applications, TensorFI [8]. We use our tool to evaluate the EdgeML TensorFlow library of algorithms for both intentional and unintentional hardware faults. With our work, users can estimate the worst case performance, thresholds of fault tolerance and dependability of the models they deploy. This is especially important when these models are used in safety-critical systems. Determining the extent of their reliability and resilience of the algorithms in the presence of faults can thus help better secure the system in advance.

III. Fault Model

The fault model we consider influences the design of our framework. We consider two categories of hardware faults in our work, unintentional faults and intentional attacks. Let us separately consider the fault model for each of the faults we emulate in our software framework:

A. Modeling unintentional hardware faults

Unintentional hardware faults can be caused by cosmic particle strikes. The error could propagate and trigger static faults in memory. This can cause changes in either the stored model parameters, or the activations and hidden states during computation [20]. We model them as bit flips during the execution of TensorFlow program with the faults abstracted to the interface level of APIs that handle the vulnerable model parameters. We assume that these faults do not modify the structure of the TensorFlow graph, and that the inputs provided into the program are correct, because such faults are extraneous to TensorFlow and are outside our scope. Finally, we assume that the faults do not occur in the ML algorithms. This assumption is needed for us to compare the output of the FI runs with the golden runs, in order to determine the accuracy difference or if a Silent Data Corruption (SDC) has occurred. We only consider faults during the inference phase of the ML program. While training is time-consuming, it is a one-time process and the results of the trained model can be checked. Inference, on the other hand, is executed many times over and over again with different inputs, and is hence likely to experience faults. This fault model is in line with other work in this area [22] [23] [24]. We discuss the new specific fault injectors developed for the EdgeML models in the implementation section.

B. Modeling intentional hardware faults

Intentional hardware faults can occur through side channels with timing information or electromagnetic emanation [19] or through software with rowhammer attacks [21]. An attacker capable of performing specific memory access patterns (at DRAM-level) can induce persistent and repeatable bit corruptions from software. Vulnerable parameters are found to be larger objects (greater than 1MB) that are page aligned allocations such as weights, biases in different layers. This is visualized in a 2 layer neural network shown in Fig. 1. We model these errors as static injections during the inference stage, as once the ML model is trained, the model parameters are known and stored so it is possible to manipulate them. We abstract the faults to the interface level of the model parameters or functions. We assume faults cannot modify the graph and that the inputs provided to the program are correct. These assumptions and fault model is in line with work in this area [17] [20]. Since the fault injections are different from what TensorFI and other frameworks can model currently, we design these different software fault injections in our new framework and call it TensorFledge.

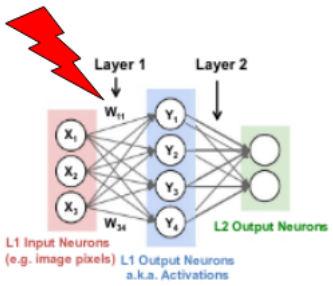

Fig. 1. Static memory faults can cause errors in the stored weights

IV. Design and Methodology

In this section, we discuss the design of TensorFledge, followed by the EdgeML model fault points that influence the design. We then provide a high level overview of the methodology employed to implement the design.

A. Design Goals

We outline the following goals in our framework design:

- **Comprehensive:** While it is not possible to model all the different types of software and hardware faults, the injector should be able to capture the fundamental artifacts of possible fault points that can occur in the program. This would allow the user to evaluate overall error resilience of the model and improve upon the vulnerable paths.
- **Flexible:** Our tool should be easy to use and customize without version or third-party library dependencies. This would allow the user to configure the fault injections according to their dependability assessment requirements.
- **Fast:** Injecting faults into the graphs or model always incurs overheads in memory and time. The injector should be able to modify the program and program states in an efficient manner so as to keep the injection cost as low as possible. This would allow the user to perform multiple runs or high density injections reasonably fast for even larger models.

B. TensorFledge Design

We satisfy the design goals outlined earlier by

- Identifying the probable fault points in the EdgeML models. These are the **comprehensive** representations in a program, called *injection artifacts*, which we modify for software fault injection purposes.
- Exposing a single function call for the user to start the fault injections and providing a configuration file to customize the type and number of injections. This makes the tool **flexible**, easy to customize for emulating a myriad of faults and faulting probabilities.
- Extracting, changing and plugging back only the parameters specified for FI and avoiding replication of the graph or interfering with other operations. Once trained, adding in the fault injection takes nearly the same time as another inference run of the model. This gives the tool efficient and **fast** performance under various fault loads.

C. Methodology

Once the design goals are laid out, we consider the right methodology to be applied to our SFI tool for the implementation. The choice in what features are developed and how they are developed are informed by taking a look at the existing methods, their advantages/disadvantages and brainstorming new alternatives that can best help us achieve our goal.

Software fault injection can be implemented by a variety of methods answering different research questions. Following are some of the questions that influence the adoption of our current methodology:

1) FI coverage rate: If we want high coverage of the fault tolerance mechanisms, we can define an exhaustive list of all the possible faults that can occur in the model. But this would give us little benefits over a formal verification technique [25], as the main trade-off of using a SFI approach in the first place, is to provide efficient implementation with little intrusiveness. So we focus on probable faults and not all possible faults, keeping with our first design goal of comprehensivity in mind.

2) FI trigger mechanism: We need to consider the different trigger mechanisms that cause the artificially generated fault to be inserted into normal program execution. These can be

(i) time-based: FI at predetermined time intervals

(ii) location-based: FI at predetermined memory locations

(iii) execution-driven: FI takes place dynamically, which depends on the control flow.

While time-based fault injection can often easily be implemented non-intrusively, this is not the case for location-based fault injection, which is suitable for memory corruption, but impedes controlling the fault load dynamically. Execution-based fault injection allows for complex and realistic fault models, but is also not applicable to black box applications. Here we focus on location-based fault injection, as most of the intentional hardware attacks are targeted towards changing certain states in memory [19] [26].

3) FI insertion time: We can insert the fault at different stages of a program execution. SFI can take place at different injection times:

(i) Before runtime: The program is modified upfront, for instance, by using source code mutation to add faults (software bugs) to the code, as static fault injection, or

(ii) During runtime: Faults are injected during program execution, as dynamic fault injection, or

(iii) At the loading time of external components: Here, injection triggers may be the dynamic binding of external libraries or the adding of other dependencies during runtime.

We choose inserting faults before runtime, as we do not want to interfere with the program run which can lead to higher overheads. This is in line with our third design goal of speed. Our second design goal of flexibility and ease of use would be violated if we choose to insert faults with external components as it adds dependencies on external libraries. We want to avoid this as much as possible.

V. Implementation

In this section, we start with a brief overview of the features in TensorFlow followed by the features we support in TensorFledge. We then dive into a discussion on the specific fault injectors for the EdgeML models to evaluate the error resilience in the presence of hardware faults. We follow up with the translation to technical implementation. Alternate design choices, limitations and other perspectives gained over the course of this work are discussed later in the Open Problems & Future Work section.

Before delving into the details, it is important to note that TensorFlow, or at least the TensorFlow v1.15 that we use and that the EdgeML models are built with, runs its programs as a computational dataflow graph. The ML operations are the nodes and the data that flows through them, the edges.

In TensorFledge, we implement FI considering faults at different points in the ML models. In general, there are four fault points identified in a typical DNN. These are weights, activations, states and datapaths [20]. Faults in weights are manifested as model-level mutations and faults in the other three are manifested as data-level mutations in a program.

Model-level mutations include changes to the underlying model such that future inputs produce faulty predictions. For example, static memory faults can change the stored training weights [21] in a DNN. In our EdgeML algorithms, the model parameters differ and are not exactly weights. They can be projection matrices, prototype scores or RNN cell parameters. They are equivalent to weights because in both cases, these parameters are vulnerable to static memory faults as they are stored model parameters from the training stage. We call these corresponding parameters "artifacts" and identify the vulnerable ones so that we can inject single bit flips or random faults into them and evaluate the resilience.

A. Identifying injection artifacts in the EdgeML models

In general DNNs, it is straightforward to identify the vulnerable parameters, as they are usually the weights and biases that are stored after training. But in the ML algorithms used for edge computing, they are not that apparent as these models do not follow the typical structure of neural networks.

For example, in ProtoNN, the kNN based algorithm, there are three trained model parameters that are learnt together for the training data. This joint learning of sparse matrices is done to provide good accuracy with decreased model size. So we inject faults into this learnt sparse matrix called the prototype projection matrix, W, the parameter representation matrix, B and the corresponding score vector, Z. These are equivalent to the function of layered weights and biases in a neural network.

Let us see another example in RNNs. Unlike feed forward networks, recurrence in RNN means the same parameters are shared across all layers. These are the RNN model parameters W and U, stored for each RNN cell after training; which we identify as the injection artifact. Similarly, we have identified the vulnerable model parameters specific to each EdgeML model and provided them in Table I.

EdgeML model	Injection artifacts
ProtoNN	Prototype projection matrix W Parameter representation matrix B Prototype label score vector Z
Bonsai	Node predictor parameters W & Z Node branching function parameters T Sparse projection matrix Z
FastGRNN	W parameter of FAST GRNN cell U parameter of FAST GRNN cell
FastRNN	W parameter of FAST RNN cell U parameter of FAST RNN cell
UGRNN	Two sets of W parameters Two sets of U parameters
GRU	Three sets of W parameters Three sets of U parameters
LSTM	Four sets of W parameters Four sets of U parameters
EMI-RNN LSTM	LSTM kernel parameters Secondary linear classifier weights W
EMI-RNN FastGRNN	Two sets of W parameters Two sets of U parameters Secondary linear classifier weights W

TABLE I
INJECTION ARTIFACTS IN EACH EDGEML MODEL

These are also provided in the YAML files along with the corresponding model descriptions in the `confFiles` directory in the TensorFledge repo.

B. Modeling injection and injector types for different faults

We model static injection for the chosen artifacts. These are collected from the model after training. During fault injection, certain values in the tensor are modified as per the configurations specified. Inference is then run again with the new faulty artifact values (an example is shown in the next section). Below is the core fault injection code:

```
1  v = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES)
2  v_ = tf.scatter_nd_update(v, i[] , f[])
3  sess.run(tf.assign(v, v_))
```

In line 1, the function `tf.get_collection()` collects the trained model parameters from the graph and stores them in tensor `v`. In line 2, `tf.scatter_nd_update()` updates certain values in `v` according to the tensor indices (`i[]`). Faults (`f[]`) are the updates to be made at the indices. `tf.assign()` updates the actual model weight tensor.

We model three types of fault injectors:

(i) Shuffle: This injector takes the stored model parameters and shuffles the tensor values randomly.

(ii) Mutate: This injector introduces single bit flips into the tensor values. The number of injections can be varied.

(iii) Zeros: This injector updates the artifact tensor values to zeros. The percentage of injection can be varied.

Shuffle is a random value replacement to see how the model fares in the presence of random faults. *Mutate* is the main injector that helps us evaluate the models in the presence of single bit flips in tensor values, which closely follows emulating the hardware faults that affect static memory. Since our EdgeML models rely on sparse matrices to keep the model size low, we want to see the effect of even sparser matrices on accuracy. This is achieved with the *zeros* injector.

C. Modeling single bit flips at different positions

We model single bit flips in the injection artifacts with the help of the *mutate* injector. We allow the exact number of bit flips to be specified in the YAML file, so users can see the effect of a bit flip in one tensor value vs bit flips in multiple tensor values, according to their needs.

Taking inspiration from BinFI [24], we also allow the bit position to be specified, so users can identify the safety critical bits in their evaluation. We find that the model parameters are stored as single precision floats (`tf.float32`). This means there are 32 bit positions a user can choose from. It is possible to see the effects of a sign bit flip or mantissa bit flip.

In the case the user does not want to specify a particular bit position, we have a random bit position be selected automatically for injections to observe the effects of a random single bit flip as well.

VI. EXAMPLE USAGE

In this section, we explain how a developer would use TensorFledge in an EdgeML application. For the example, we consider the ProtoNN model. ProtoNN [1] is a novel kNN based algorithm for tiny devices and one of the EdgeML models. Before explaining the usage, we provide a brief study on the differences of a general kNN algorithm and ProtoNN.

A general kNN works by finding the distances between a query and all the examples in the data, selecting the specified number examples (k) closest to the query, then voting for the most frequent label (in the case of classification) or averaging the label (in the case of regression). This general kNN is not suitable for the IoT setting because it requires bigger model sizes than tiny devices can hold. The entire training data has to be used for any reasonable accuracy, which is not favorable to do because of power and memory constraints in IoT devices. In addition, kNNs require computing the distance of a given test point wrt each training point, which takes a long time and is simply prohibited for fast prediction in real-time settings. ProtoNN addresses these by using a sparse low-d projection matrix (low model size) that is jointly learned with prototypes that represent the entire training data (lesser time). Additionally, labels for each prototype are learnt which boosts accuracy and allows extension for multi-label problems.

Now we explain the usage. First, the user has to identify the injection artifact. The injection artifacts for ProtoNN are the prototype projection matrix W, the parameter representation matrix B and the prototype label score vector Z. The chosen artifact needs to be specified in the YAML file. Next, the injector has to be chosen. Let us say the user wants to modify W and observe the effect of a random single bit flip in W. The YAML file for this configuration would be:

```
# ProtoNN.yaml
Artifact: 0
Type: mutate
Amount: 1
Bit: N
```

We specify '1' in 'Amount' for a single bit flip, and 'N' in 'Bit' to indicate we want the bit position chosen randomly.

Next, we need to use the TensorFledge APIs in the ProtoNN example. For brevity, we omit the training code in the ProtoNN example (complete code found here) and jump to the point of injection. We need to call the `inject` method to perform the injections. We first need to import the module and then add the call with the current TF session as the argument. This is shown below:

```
1   import inject
2   ...
3   # Print some summary metrics
4   acc = sess.run(protoNN.accuracy, feed_dict={X: ↘
        x_test, Y: y_test})
5   print("Accuracy before injections: ", acc)
6   inject.inject(sess)
7   acc = sess.run(protoNN.accuracy, feed_dict={X: ↘
        x_test, Y: y_test})
8   print("Accuracy after injections: ", acc)
9   ...
```

In line 1, the inject module of TensorFledge is imported into the file. In line 6, the call `inject(sess)` starts the injection according to the configuration specified in YAML. The result is shown in Fig 2.

```
Accuracy before injections:  0.88739413
/ubc/ece/home/kp/grads/nniranjhana/TensorFledge/inject.py:2
1: YAMLLoadWarning: calling yaml.load() without Loader=...
is deprecated, as the default Loader is unsafe. Please read
 https://msg.pyyaml.org/load for full details.
  fiConf = yaml.load(fiConfs)
WARNING:tensorflow:From /ubc/ece/home/kp/grads/nniranjhana/
TensorFledge/inject.py:50: The name tf.get_collection is de
precated. Please use tf.compat.v1.get_collection instead.

Accuracy after injections:  0.88689584
```

Fig. 2. Injection accuracy difference for one random bit flip in ProtoNN

As we can see, the accuracy difference is 0.05% for a single random bit flip in the projection matrix W of ProtoNN model. This is expected because W is of dimensions (256, 60). Thus changing one bit in one tensor value is not going to cause a huge variation in how the model predicts as it relies on all the other uncorrupted parameters as well. We will see more variations in accuracy with other fault configurations and fault probabilities. We examine these in detail in the following evaluation section.

VII. Evaluation

In this section, we first define our evaluation metric followed by the research questions the evaluation is based on. We then describe our experimental set up. We follow up with the evaluation results for different fault types and fault probabilities in the next section.

A. Evaluation Metric

The evaluation metric we consider is the relative accuracy injection difference (RAID). This is defined as the percent difference in accuracy before and after injections. As the starting accuracy for various EdgeML models is different, we need to consider the relative change. An alternate evaluation metric that is usually considered is the Silent Data Corruption (SDC) rate. The reasoning behind why the RAID metric was better suited for our use cases is discussed in the Open Problems & Future Work section.

B. Research Questions

We ask the following research questions:

RQ1. What are the RAID rates for the EdgeML models under random shuffle faults?

RQ2. What are the RAID rates for the EdgeML models under random bit flip faults?

RQ3. What are the RAID rates in a single EdgeML model under different specified bit flip positions?

RQ4. What are the RAID rates for the EdgeML models with more sparse model parameters?

RQ5. What are the overheads incurred during the static fault injection?

C. Experimental Setup

Hardware: All our experiments were conducted on nodes running Ubuntu 16.04 with Intel Xeon 2.20GHz processors with 12 cores and 64 GB memory. Note that since we use relative percentages or ratios in our experimental results, the actual hardware configuration does not matter.

Software: We use TensorFlow v1.15 and Python v2.7. We use the current version of the EdgeML models (latest commit 9dd5843). All the other software dependencies and their versions are outlined in the installation guides.

ML Models: We use the sample examples provided for the EdgeML models. We evaluate 8 of the 9 EdgeML models written using TensorFlow. These include the ProtoNN, FastGRNN, FastRNN, UGRNN, GRU, LSTM, EMI-LSTM and EMI-FastGRNN models. We were not able to evaluate the Bonsai example as the model defined in EdgeML has not separated the training and testing phases. Different RNN models like the UGRNN, GRU, LSTM, FastRNN, FastGRNN are evaluated by calling the FAST algorithm code with these different RNN cells using the `-c` option listed in the accompanying file `helpermethods.py`. EMI-RNN is an approach that can be applied to multiple RNNs. E stands for early prediction and MI for multi-instance learning. So we can consider EMI-RNN applied to LSTM or FAST cells. We construct a couple of the models from pieces of code provided in Jupyter notebooks. These are the EMI-RNN examples (the EMI-LSTM and EMI-FastGRNN Python code).

ML Datasets: Most of the models use the USPS-10 dataset [27]. This is a handwritten digits dataset much like the MNIST, but on a smaller scale suited for IoT. For example, the image dimensions here are 16x16 pixels and not 28x28 like in MNIST. There are 7k training images and 2k test images, much smaller than the 60k training images and 10k test images of MNIST. There are again 10 labels. For RNN models, even though USPS-10 is not a time-series dataset, it can be assumed as one where each row is coming in at one single time. So the number of time steps would be 16 and the input dimensions fed to the RNN will also be 16. All the models except the EMI-LSTM and EMI-FastGRNN use the USPS-10 dataset. These two models use the HAR (human activity recognition) dataset [28] as this multivariate dataset is better suited to make use the early prediction algorithm in EMI.

FI Settings: We perform only 500 injections for each of the experiments because of time constraints. We extract the average obtained in the accuracy post injection. We usually consider W as the injection artifact. This corresponds to the projection matrix in ProtoNN and the first set of RNN cell parameters for the other networks. Experiment specific configurations are mentioned along with the corresponding experiment results in the next section.

VIII. Results & Discussion

We use different configurations available in TensorFledge to answer the Research Questions posed in the previous section. We organize the results by the same:

RQ1. What are the RAID rates for the EdgeML models under random shuffle faults?

To answer this RQ, we consider the common injection artifact across the models and apply random shuffle on it. This corresponds to W, the first parameter stored after training each model. Here, we explicitly show the actual accuracy values before and after injections for easier visualization. For the coming experiment results, we would only show the percent RAID. The results are shown below in Fig 3.

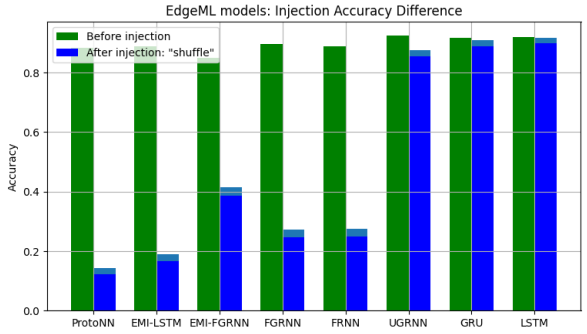

Fig. 3. RAID rates for random shuffle faults. Higher blue bars are better. Error bars range from $\pm 0.7583\%$ to $\pm 1.455\%$ at the 95% confidence interval.

As we can see, the more error resilient models by this fault type are UGRNN, GRU and LSTM. This is because these models have two, three and four sets of W and U parameters respectively that are learnt for the RNN cell. Whereas 4 of the other models excluding EMI-FGRNN have only one set of W and U parameters. Since we modify only one W set, the prediction remains unaffected as it relies on the remaining sets. We can see that from UGRNN, just adding one more set of these parameters for the RNN gives tremendous increase in error resilience. EMI-FGRNN has two sets as well, but in addition it has a secondary linear classifier, which by the nature of the EMI-RNN algorithm heavily relies on the parametrized W and U matrices, and thus the error in W is compounded. Thus it is not able to achieve as high accuracy rates as the other three which solely rely on the individual RNN cell parameters. FGRNN and FRNN have better accuracy than ProtoNN and EMI-LSTM because in addition to W, they have a U set that defines the learnt RNN cell.

RQ2. What are the RAID rates for the EdgeML models under random bit flip faults?

For this experiment, we perform bit flips at randomly chosen positions for a number of values in the chosen injection artifact. The bit flips occur in multiples of 5 starting from 1 to 25. We chose only upto 25 number of bit flips because of two reasons. Firstly, it is highly unlikely that there occurs a large number of bit flips, so we stick to a lesser number as it more closely models the real world scenario. Second, to observe significant changes in accuracy and to see which models are better resilient under higher fault loads, we see the effects of change over a range of faulting probabilities. The results for this experiment are shown below in Fig 4.

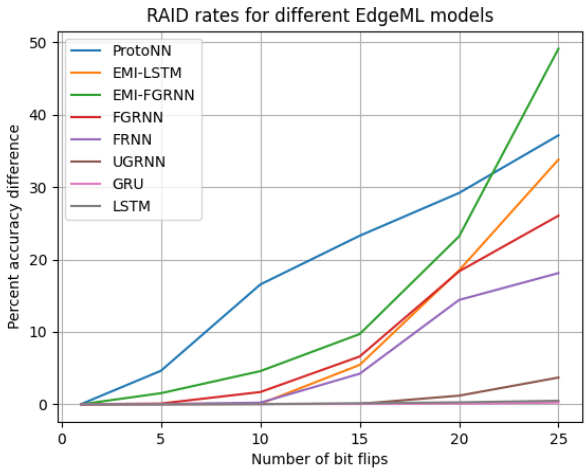

Fig. 4. RAID rates for random bit flips. Less growth is better.

We see that the UGRNN, GRU and LSTM models are once again the most resilient under random bit flips too. This is because of the same reasons we ascribed to earlier. Next, we see that the EMI-LSTM and EMI-FGRNN have nearly an exponential rate of decrease in accuracy. This could be because the EMI-RNN algorithm uses the multi-instance learning method where the sequential data points are split into a collection of instances and are assigned it's own class label. Some instances may contain the true class, some will not and are noisy, negative labels. When there are more bit flips, changes in W causes changes in these negative labels as well and they end up carrying more weightage in the decision. Thus we see a drastic change in accuracy with more bit flips. FRNN and FGRNN follow similar patterns throughout as the only real difference between them is the gated RNN cell in FGRNN, other than that, the model and model parameters are quite equivalent. ProtoNN can be considered the least resilient even in the presence of low faults and has a linear increase. This is because it is a forward network and so more faults in weight projection matrix leads to more degradation in accuracy. We also note that the standard deviation in accuracy rates for random bit flip faults was found to be much larger than *shuffle*, or with *zeros* injectors.

RQ3. What are the RAID rates in a single EdgeML model under different specified bit flip positions?

For this experiment, we choose the FastGRNN model as we see from previous experiments that it seems to show the average performance in error resilience compared to other models. We also set the amount of bit flips to be 15, as we observe significant changes at this stage, and also it is in the middle of previously set numbers from 1 to 25. We conduct fault injections at different bit positions, spanning 0 to 31. We define the 'safety critical bit' to be the bit position that causes the most change in accuracy and we find it for this model. The results for this experiment are shown below in Fig 5.

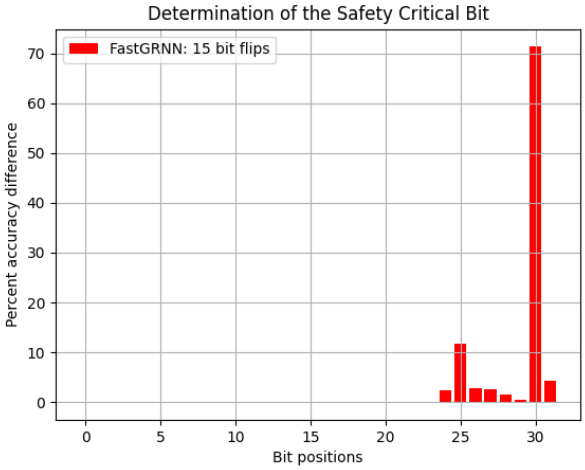

Fig. 5. RAID rates for 15 bit flips at different bit positions. Bit position 30 is determined to be the safety critical bit for the FastGRNN model.

To understand the results, first we take a look at the single precision floating point number format shown below in Fig 6.

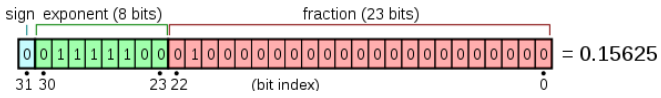

Fig. 6. Single precision floating point format

We can see that the first 23 bits are used to represent the significand. Thus we don't see any changes in accuracy until we reach the bits where the exponents are stored. We can see about 2% change in the accuracy in the 24th bit. The safety critical bit is the bit that causes drastic degradation in accuracy thus making the prediction meaningless. This is identified as the 30th bit in the W matrix for the FastGRNN model. Since this bit represents the most significant digit of the exponent, this could mean a tensor value changing from 0.18298629 to 6.22670076244e+37. This explains the huge change in accuracy. The second largest change in accuracy is observed at the 25th bit. This represents the most change in the significand (an example taken from the experiment: a number changing from -0.09936746 to -1.58987939358). We see a gradual decrease in accuracy change until the 30th bit. The 31st bit is the sign bit, and we see that it constitutes the third largest change. A number changing from -0.055914197 to 0.0559141971171 (again, from the experiment) does not create a huge change since the model parameters are small in the FastGRNN model. So the change is centered around 0. This will obviously not hold for larger and different models.

RQ4. What are the RAID rates for the EdgeML models with more sparse model parameters?

For this experiment, we use the *zeros* injector with different percentage of zeros in the injection artifact. One common aspect between all the EdgeML models which is not found in conventional ML models is that the model parameters are represented as sparse matrices to reduce model size and save up on storage. So we evaluate how much the accuracy degrades with even more sparse matrices. We start with 1 % and go up to replacing 75% of the artifact tensor values with zero and judge which model performs best. The results for this experiment are shown below in Fig 7.

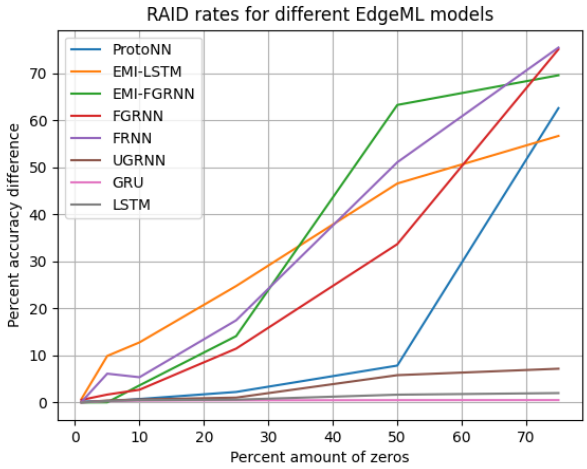

Fig. 7. RAID rates for percent sparse matrices. Less growth is better.

As we can see, UGRNN, GRU and LSTM are once again the most error resilient. But even they experience upto 8% accuracy degradation when 75% of one of their model parameters are zeros. ProtoNN shows an interesting growth rate, with a sudden jump after 50% and performing similar to the other RNNs at 75%. This is because, the model parameters for the ProtoNN are by default up to 60% sparse, so replacing until 50% of it with zeros did not make much of a difference. FRNN and FGRNN show similar trends, and FGRNN seems to perform better because of lesser sparsity attributed to gated unit with weighted residual connection for every hidden state. Even though FGRNN takes more model size and more prediction time compared to FRNN, it seems to make up for it in better error resilience. EMI-FGRNN performs better than EMI-RNN LSTM for similar reasons. Even though LSTM has 3 sets of W and U parameters while GRU has only 2, GRU still performs better again because of the gated unit.

Injector	Amount of injections	Overhead (s)	Overhead (x)
Shuffle	100%	0.11	1.1
Zeros	1% - 100%	0.11	1.1
Mutate	1	0.11	1.1
	10	0.5073	4.6
	25	1.1758	10.7
	50	2.3291	21.2
	75	3.6750	33.4
	100	5.0710	46.1

TABLE II
INJECTION OVERHEAD TIME IN SECONDS AND MULTIPLICATIVE FACTOR

RQ5. What are the overheads incurred in static fault injection?

To answer this RQ, we can pick any model as we only test overheads incurred by the different injector types for different number of tensor values. We note that the baseline without injections is on average 0.1 seconds. We use Python's `time` module and measure the time difference (calling `time.time()` before and after injections) to get the injection overhead. We choose the FastGRNN model again, and display the values obtained in Table II.

We see that the *shuffle* and *zeros* injector incur minimum overheads. The overheads for the *mutate* injector increase with the number of injections. This is because for each tensor value, the `bitflip()` function is called, which has 5 operations on its own. So the cost increases a lot with more number of bit flips. But since we can't possibly require to flip the bits of more than 100 values, the maximum time taken of 5 seconds is acceptable.

IX. Open Problems & Future Work

In this section, we discuss alternate design choices we considered earlier, implementations and experiments that did not work out, the limitations of our current implementation and possible future work directions. This section can also be treated as an informal and technical lessons learnt section as we derive new perspectives from failures.

On the extension of TensorFI: A natural question to ask is if there already exists a fault injector for general TensorFlow applications, why not use it or extend it for these EdgeML programs as well? The short answer is honestly, we tried.

Our motivation is a little different, but agreed that with the usage of TensorFI, we could have achieved the datapath mutation mentioned earlier. So initially, we tried to instrument the existing EdgeML example code with TensorFI, but ran into many errors. With prior experience, we reasoned this out to be because of certain operators used in EdgeML not having their corresponding fault injector implemented in TensorFI. And so we implemented/modified the following:

- `injectFaultTranspose` for the Bonsai model
- `injectFaultTensorDot` for the `tf.tensordot` function in the EMI-RNN model
- `injectFaultReduceSum` for the `tf.reducesum` function in the ProtoNN model
- `injectFaultUnstack` for the RNN models (FastRNN, FastGRNN & EMI-RNN)

The instrumentation then worked. But fault injection still did not. On deeper inspection, we found that the fundamental way in which TensorFlow was building the graph for the EdgeML models was different and TensorFI could not completely replicate it as-is. Some obvious differences were that the number of operations itself was different in each. It is difficult to debug when there are 2860 operations. But a lot of it had to do with TensorFI replicating the node based on "type" of the operation alone. So in the EdgeML models, some operations are grouped and have the same type but do different steps of a whole operation. These are treated as separate operations in TensorFI and so is not able to handle the input correctly. For comparison, I have added both the original TensorFlow graph and the FI map generated by TensorFI for the corresponding nodes for the Bonsai model here.

On dynamic injection: One of our starting goals was to also model faults in activations and hidden states. Activations and hidden states are input-dependent, dynamic values. Injecting faults into these units requires changes to the TensorFlow inference computation. Activation and state fault injection operators can be implemented as GPU-compatible element-wise tensor operations. For GRU layers, we can inject faults into hidden states at each time step between state updates. The network can then be evaluated in the configured fault environment. Modeling activation and state fault injection requires eager execution and the new `tf.function` introduced in TF v2.0. Current EdgeML models are yet to be ported to TF v2.0. So while implementing them is certainly possible (and in progress as part of my research), we cannot get any experimental results for the EdgeML models without TF v2.0 support.

Why not SDC as the evaluation metric? The difference between EdgeML models and general ML models are that since time and space is limited, model parameters are jointly learnt across all training data. So observing a single input and if it affects the output is less meaningful than testing the final accuracy across multiple inputs. This is why we prefer accuracy. Since we measure the relative accuracy difference, the initial testing accuracies do not matter and will not bias our evaluation. But mostly they were in the 88% to 90% range.

Future work: The evaluation can have more extensions. For example, in a single model, effects of different model parameters considered for injection artifacts can be studied. Considering the effects of larger amount of bit flips for the more resilient networks can be studied. We set the parameters after limited initial checks, and so could not redo them after 500 injections due to time constraints.

One can use the tool to find the safety critical bits with respect to other parameters and in the other EdgeML models as well. It might prove useful to study the variance between them, if any.

Another direction is considering the RAID rates in a single EdgeML model under different hyperparameter configurations. This might answer the question if the user is choosing the optimal hyperparameters when error resilience is also considered.

X. Conclusion

In this project, we build a fault injector for the EdgeML TensorFlow library called TensorFledge. We explore the design goals and explain the usage and implementation of our tool. We use the tool to evaluate 8 EdgeML models in the presence of static memory faults in hardware that could cause changes in the model parameters. We perform experiments with different fault types and faulting probabilities and discuss why some models are more error resilient.

We find that UGRNN, GRU and LSTM are the most error resilient models, with accuracy changes of less than 8% even under high fault loads. We also use our tool to find the safety critical bit in the FastGRNN model parameters. Since the ML models used in edge computing tend to be small and deal with sparse model parameters, we observe the effects of sparsity on error resilience and find that having a gated unit increases performance in this context. Finally, we address some problems faced during the course of this project and provide some hints for future research directions.

References

[1] C. Gupta, A. S. Suggala, A. Goyal, H. V. Simhadri, B. Paranjape, A. Kumar, S. Goyal, R. Udupa, M. Varma, and P. Jain, "ProtoNN: Compressed and accurate kNN for resource-scarce devices," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. International Convention Centre, Sydney, Australia: PMLR, 06–11 Aug 2017, pp. 1331–1340. [Online]. Available: http://proceedings.mlr.press/v70/gupta17a.html

[2] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 KB RAM for the internet of things," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. International Convention Centre, Sydney, Australia: PMLR, 06–11 Aug 2017, pp. 1935–1944. [Online]. Available: http://proceedings.mlr.press/v70/kumar17a.html

[3] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 27–38.

[4] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, and et al., "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, p. 129–173, May 2014. [Online]. Available: https://doi.org/10.1177/1094342014522573

[5] "Row hammer attack," https://en.wikipedia.org/wiki/Row_hammer.

[6] J. Carreira, H. Madeira, J. Silva, and D. Informtica, "Xception: Software fault injection and monitoring in processor functional units," *Proceedings of the 5th IFIP Working Conference on Dependable Computing for Critical Applications*, 03 2001.

[7] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using fiat," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 575–582, April 1990.

[8] G. Li, K. Pattabiraman, and N. DeBardeleben, "Tensorfi: A configurable fault injector for tensorflow applications," 10 2018, pp. 313–320.

[9] "Microsoft EdgeML library," https://github.com/microsoft/EdgeML.

[10] https://en.wikipedia.org/wiki/Machine_learning#Applications.

[11] https://www.microsoft.com/en-us/research/project/edgeml/.

[12] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," *CoRR*, vol. abs/1901.02358, 2019. [Online]. Available: http://arxiv.org/abs/1901.02358

[13] D. K. Dennis, C. Pabbaraju, H. V. Simhadri, and P. Jain, "Multiple instance learning for efficient sequential data classification on resource-constrained devices," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 10976–10987.

[14] Y. Cherapanamjeri, P. Jain, and P. Netrapalli, "Thresholding based efficient outlier robust pca," 02 2017.

[15] L. Feinbube, L. Pirl, and A. Polze, "Software fault injection: A practical perspective," 2017.

[16] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore," *Proceedings of the 26th Symposium on Operating Systems Principles - SOSP '17*, 2017. [Online]. Available: http://dx.doi.org/10.1145/3132747.3132785

[17] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepmutation: Mutation testing of deep learning systems," *CoRR*, vol. abs/1805.05206, 2018. [Online]. Available: http://arxiv.org/abs/1805.05206

[18] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Comput. Surv.*, vol. 51, no. 1, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3143561

[19] J. Breier, D. Jap, X. Hou, S. Bhasin, and Y. Liu, "Sniff: Reverse engineering of neural networks with fault attacks," 2020.

[20] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, June 2018, pp. 1–6.

[21] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitras, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 497–514. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/hong

[22] G. Li, S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," 11 2017, pp. 1–12.

[23] M. Sabbagh, C. Gongye, Y. Fei, and Y. Wang, "Evaluating fault resiliency of compressed deep neural networks," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, June 2019, pp. 1–7.

[24] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "Binfi: An efficient fault injector for safety-critical machine learning systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356177

[25] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Towards proving the adversarial robustness of deep neural networks," *Electronic Proceedings in Theoretical Computer Science*, vol. 257, p. 19–26, Sep 2017. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.257.3

[26] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Deeplaser: Practical fault attack on deep neural networks." 2018.

[27] https://www.kaggle.com/bistaumanga/usps-dataset.

[28] https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones.