# Fine-Grained Vulnerability Analysis of Resource Constrained Neural Inference Accelerators

Panayiotis Corneliou, Panagiota Nikolaou, Maria K. Michael, Theocharis Theocharides
Department of Electrical and Computer Engineering and
KIOS Research and Innovation Centre of Excellence, University of Cyprus

***Abstract*—The use of Neural Network (NN) inference on edge devices necessitates the development of customized Neural Inference Accelerators (NIA) in an attempt to meet performance and accuracy requirements. However, edge infrastructure often relies on highly constrained resources with limited power budget and area footprint. At the same time, reliability is very crucial, especially, for critical applications and trade-offs area and power due to the needed for protection. In this paper, we study the soft-error vulnerability of an edge NIA, using an emulation-based fault injection framework, which allows for accurate and fine-grained analysis. We consider the tinyTPU architecture, which resembles Google's Tensor Processing Unit (TPU) but is optimized for edge-based applications. Through a proposed error outcome taxonomy for NN-based algorithms, we study the criticality of each NIA component and explore their vulnerability to Single Event Upsets (SEU), while providing analysis of performance-accuracy trade-offs such as using smaller NN models and periodic memory refresh. Further, through analysis of the tinyTPU architecture, we manage to considerably reduce the emulation time for components with non-persistent faults.**

***Index Terms*—Artificial Neural Networks, Vulnerability Analysis, Fault Tolerance, Single Event Upsets, Soft Errors, Edge Accelerators, Emulation-Based Fault Injection**

## I. Introduction

Neural Networks (NNs) have been applied in multiple safety critical applications, such as wearable biomedical diagnostics and unmanned aerial and terrestrial vehicles [1], [2]. Typically, these applications use dedicated hardware accelerators, embedded in edge devices to process the inference with low communication latency. Despite the substantial advantages, edge devices are resource constrained due to limited physical and power footprint. Thus, several techniques have been introduced to reduce the size of the NNs in order to utilise less hardware and power with minimal impact on accuracy [3]. Additional reduction of NNs accuracy can be induced by soft errors, caused by power supply voltage-scaling or environmental conditions [4]. These errors can either cause application or system crashes, or affect the program's output (silent data corruption errors), leading to misclassifications that in some cases can be catastrophic [5].

Existing redundancy techniques for soft error mitigation come with high cost on area, power, and performance, which edge devices cannot afford [6]. Consequently, it is important to explore the vulnerability of these devices at design time and apply selective redundancy at a fine-grained level, as needed [7]–[10]. Recent works that study the vulnerability in the presence of soft errors in various accelerators are reviewed in [4]. These works are based on either neutron beam experiments on real platforms [7], [11], or fault injections using either high-level simulation [12]–[14], or emulation [15]–[19]. This paper extends our previous work in [18], [19], an emulation-based fault injection framework, to provide fine-grain vulnerability analysis of a resource constrained edge accelerator architecture. The vulnerability analysis is performed by using an error outcome taxonomy, which categorizes the impact of Silent Data Corruption (SDC) errors based on the NN classification outcome between the fault-free and faulty NN. Particularly, in this analysis we consider the tinyTPU [20], a systolic array architecture geared towards edge devices that resembles Google's TPU [21], using the MNIST dataset [22].

Via our analysis process we, also, identify the tinyTPU components with non-percistent faults, i.e, faults which only propagate for a limited number of cycles, due to the tinyTPU architecture. This allows us to reduce the emulation time per fault injection in these components, and achieve a considerable speed-up compared to baseline approach that does consider non-persistent faults. The overall methodology can be easily extended to apply alternative accelerator architectures and NN inference algorithms.

The rest of the paper is organized as follows: Section II gives information on the architecture of the edge tinyTPU accelerator. The considered vulnerability evaluation methodology is presented in Section III, which includes the underlying SDC classification-based taxonomy and an overview of the fault injection mechanism. Section IV discusses the experimental use case and setup. The vulnerability analysis for each component is presented in Section V, and then a more fine-grained analysis is performed for all the components for different NN models. Section VI concludes this work and gives future directions.

## II. Edge Accelerator Architecture (tinyTPU)

TinyTPU architecture consists of ten main components, 41,861 registers and around 4.4M bits as shown in Table I. Figure 1 illustrates the accelerator architecture which comprises of: (1) Unified Buffer (UB): a read/write memory array that stores the input and output data of all the layers, (2) Systolic Array Control (SAC): an array that transforms data derived from UB, (3) Control Weight ($WB_{Ctrl}$): responsible for the read process of WB, (4) Control MMU ($MMU_{Ctrl}$): controls the data transferred to and from MMU, (5) Control Activation ($ACT_{Ctrl}$): enables activation component if needed, and selects the appropriate activation function, (6) Control Coordinator ($COORD_{Ctrl}$): decodes step-by-step instructions

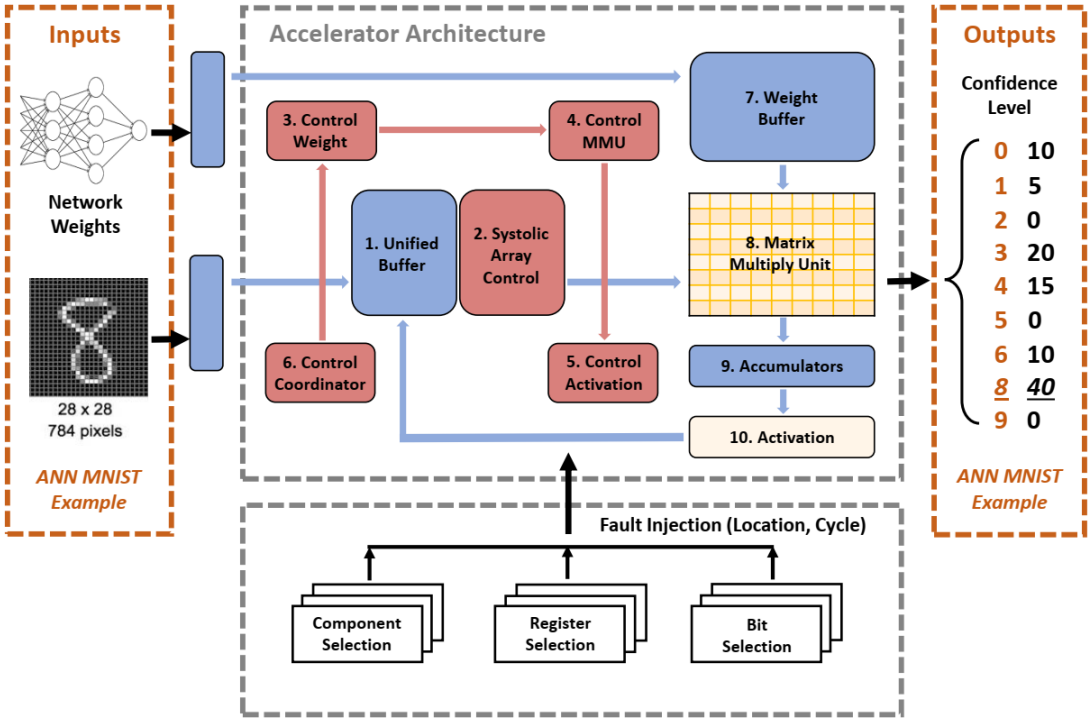


Fig. 1: Considered framework, including tinyTPU Architecture and Fault Injection Emulation Mechanism

TABLE I: Total Number of Registers/Bits of tinyTPU

| Component | # Registers | # Bits |
|---|---|---|
| 1. Unified Buffer (UB) | 4153 | 459424 |
| 2. Systolic Array Control (SAC) | 91 | 728 |
| 3. Control Weight ($WB_{Ctrl}$) | 17 | 83 |
| 4. Control MMU ($MMU_{Ctrl}$) | 21 | 435 |
| 5. Control Activation ($ACT_{Ctrl}$) | 17 | 1253 |
| 6. Control Coordinator ($COORD_{Ctrl}$) | 6 | 85 |
| 7. Weight Buffer (WB) | 32825 | 3670688 |
| 8. Matrix Multiply Unit (MMU) | 982 | 13847 |
| 9. Accumulators (ACC) | 2674 | 234953 |
| 10. Activation (ACT) | 75 | 1298 |
| **Total** | 41861 | 4382794 |

of NN inferences and provides control signals throughout the system accordingly, (7) Weight Buffer (WB): a read/write memory array that stores the training weights, (8) Matrix Multiply Unit (MMU): a two-dimensional systolic array of Multiply and ACcumulate (MAC) units that multiplies the data derived from SAC with the weights from WB, (9) Accumulator (ACC): stores and accumulates the multiplied results derived from MMU, (10) Activation (ACT): adds non-linearity to the results derived from ACC using activation functions; its output is stored back to the UB for further processing, if needed. In Figure 1, interactions between the controllers and the other components are omitted for clarity purposes.

Figure 1 also illustrates how the input (for example, an image from the MNIST dataset) and the NN weights are loaded to the UB and WB, respectively. In the scenario considered in this paper, outputs are classified using confidence level (for each possible digit 0 to 9 in this example). The highest confidence level, is used to classify the input. For the particular example shown in Figure 1, the highest confidence level (40) classifies correctly that the input image is digit 8.

## III. Vulnerability Evaluation Methodology

### A. Error Taxonomy

The error outcome caused by a bit-flip can be categorized as masked error, Silent Data Corruption (SDC) error, or crash/hang error. Figure 2 shows the error outcome taxonomy used throughout this paper. Masked errors do not manifest at the final output. On the other hand, crashes/hangs cause undesirable behavior to the running application and SDCs show an output discrepancy between the fault-free and faulty runs, and can manifest differently according to their exact impact on the classification outcome (correct or not). When an SDC produces a misclassification (MC), the error can be categorized as tolerable (TMC), critical or no impact error. TMC errors occur when we have different outputs (i.e., different confidence levels) between the fault-free and faulty runs, however, the result of the classification remains exactly the same. No impact errors are similar to TMC errors (i.e., they are also tolerated), with the only difference that the result of the misclassification changes to a different, incorrect output. For example, in Figure 1, a no impact error happens when input 8 is classified as 6 in the fault-free scenario and as 3 in the faulty one. In contrast to TMC and no impact errors, critical SDCs have negative impact on the classification as they result to an incorrect classification in the faulty case while there was a correct classification in the fault-free case. For example, an SDC causing the confidence level for digit 8 to drop to value 15 (from 40) will result to a miss-classification

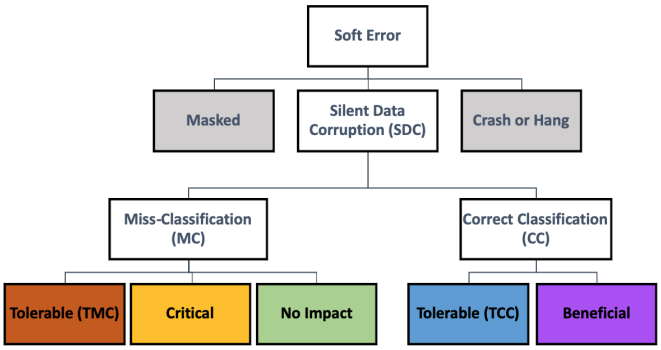

Fig. 2: Error Outcome Taxonomy

since the NN will classify digit 3 instead of digit 8. Thus, critical errors reduce the accuracy of the classification.

In the case of an SDC leading to correct classification (CC), the error can be categorized to either tolerable (TCC) or beneficial. TCC errors, as in TMC category, occur when the SDC changes the confidence level between the fault-free and faulty runs, but the final classification is not affected. As an example, let an SDC change the output(s) in Figure 1, causing the confidence level for digit 8 to drop to 35 (from 40). In this case, the output confidence level is different, but the classification remains correct. Beneficial SDCs have a positive impact on vulnerability, since they change the output of the classification from incorrect to correct. The presented taxonomy is generic and can be adopted by different classifiers, different confidence levels. Also, it can be adapted to other types of classification algorithms.

### B. Fault Injection Mechanism

The fault injection mechanism used in this paper is based on the VHDL framework in [18], [19]. This mechanism provides vulnerability analysis using hardware-based models for fault injection which are integrated with the system to be evaluated (see Figure 1). Fault injection experiments can be applied either at the entire system level or at the component and register level for finer-grained evaluation. Each granularity level has its dedicated randomization mechanism, which allows for appropriate selection on when and where to inject a fault. In each case, the impact of an error on the entire system is emulated.

Furthermore, we incorporated the following additions and optimizations, allowing for the effective execution of very large fault injection campaigns in an automated fashion, while at the same time enabling the parameterized evaluation of alternative accelerator architectures.

*1)* ***Injection of errors****:* The FI mechanism can inject single event upsets (SEU), as well as multiple event upsets (MEU). In this work we focus on SEUs experiments. Each fault injection is executed in real-time, which decreases substantially the fault campaign's overall execution time.

*2)* ***Capability of injecting faults in Memory****:* We allow fault injection in memory components, as it is necessary for studying the vulnerability of edge-based accelerators for which memory protection techniques may prove to be prohibitive due to their resource and latency constrained requirements. In particular, tinyTPU's WB is implemented in blockRAM with 112-bit length per element (tinyTPU word size). To inject a fault in WB, the targeted faulty register ID is translated to the corresponding address in the blockRAM. Next, the data from the particular address are retrieved, modified to emulate the fault, and then stored back. An extra register is used to store the fault-free value, which is used at the end of a fault injection experiment to restore the data back to the fault-free value. This allows for fast execution of several experiments, without the need of re-writing all the memory bits.

*3)* ***Parameterizable hardware design****:* We have extended the tool to be parameterizable to alternative accelerator designs. The fault injection mechanism works as an external module which takes as input the number of components, the number of registers per component and the number of bits in each register, and generates the appropriate outputs for injecting faults to the targeted design. The interface between the fault injection mechanism and the accelerator still requires some customization, nevertheless, this is an important generalization that can be used for emulation-based fault injection soft-error vulnerability analysis for a variety of architectures, including different accelerator implementations.

## IV. Experimental Use-Case and Setup

We consider the inference phase, running on the tinyTPU [20], and evaluate the vulnerability of one hidden-layer NN using the MNIST dataset. The MNIST dataset uses 28 X 28-pixel gray scale images of handwritten digits from 0 to 9 as input and includes a set of 60000 images, which we have used for the training phase to derive the needed weights, and a set of 10000 test images that we use during the evaluated inference phase [22].

Initially, an NN with 126 neurons (ANN126) within the hidden layer is considered, which uses Sigmoid activation function for both hidden and output layers. In Section V-D additional smaller NNs are evaluated.

SEUs are injected in all the 10 components of the architecture, while the inference of MNIST workload is running. To evaluate each component, the whole system's inputs and outputs are used (images and classification) - and not the isolated inputs and outputs of each component. It is important to stress out that our aim is to evaluate the architecture's components, therefore we assume that all the input data (weights, images and instructions) are fault-free, when they enter the tinyTPU.

To determine the total number of faults to be injected and ensure the statistical correctness of the experiments, we used the method introduced in [23], with a confidence level of $99\%$ and an initial error margin of $4\%$. Reducing the emulation latency as we show in Section V-A, allowed for a further reduction of the error margin to $1\%$ for all the components with non-persistent faults (all components except of the WB). Therefore, for each component with $1\%$ error margin we inject 17136 faults and for the WB we inject 1039 faults with $4\%$ error margin. We observe that the same number of

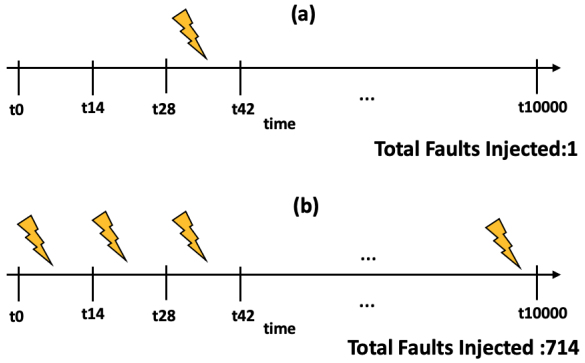


Fig. 3: Emulation Latency (a) when a fault is injected per 10000 images and (b) per batch of 14 images

fault injection experiments are needed for each component with non-persistent faults, even though they have a noticeable difference in their size. This happens due to fact that the number of needed fault injection experiments depends on both the component's size and the workload's total cycles, where the number of cycles (around 53M cycles) dominates the component's size [23]. In total, we perform 138127 fault injections for all the components.

The experiments were performed in real-time on the Xilinx Zynq-7000 SoC ZC702 evaluation board. The architecture of the tinyTPU was implemented and synthesized through Vivado, using VHDL and mapped to the FPGA via Vitis. The training phase was implemented in the host PC, using Tensorflow, and the NN weights were quantized from 32 bits-floating point to 8 bits-integer, and then transferred to the FPGA and stored in the weight buffer (WB).

## V. Experimental Results

### A. Reduction of emulation latency

TinyTPU architecture allows parallel execution of up to 14 images, leading to 714 batches for the 10000 images of the inference workload we consider (10000/14). It is observed that when a new batch of images is loaded, any previous error is removed/overwritten due to the non persistent nature of soft errors. Hence, any single fault injection can affect the process until the end of a single batch and cannot be propagated to any other upcoming batch. Based on this architectural-depended observations, we can speed-up the process by injecting one fault per batch, aggregating a total number of 714 fault injections per workload execution (i.e., accelerating the process by 714 times). In addition, through this approach we ensure that faults will be injected evenly along the 10000 images. Figure 3 illustrates the fault injection process throughout the workload's execution when we inject a fault per 10000 images and when we inject a fault per batch of 14 images. The Weight Buffer is the only component with persistent soft errors due to the fact that all NNs' weights are stored within the WB and are not overwritten until the end of the execution. Table III shows the number of cycles for 14 and 10000 images respectively.

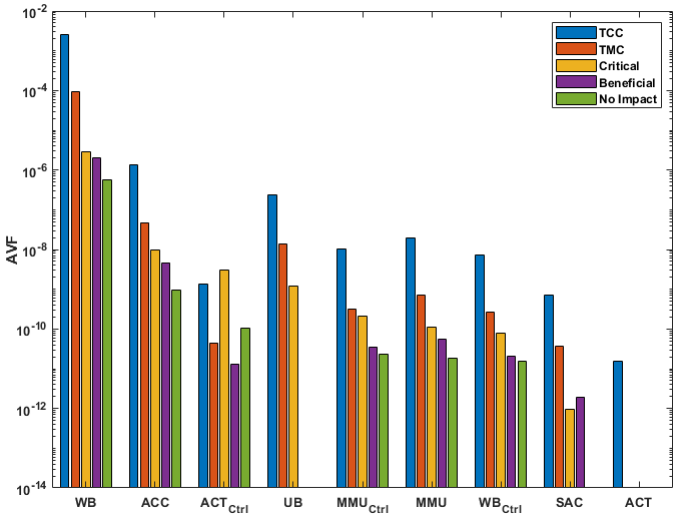


Fig. 4: AVF per Component for ANN 126

### B. Component Vulnerability Analysis

Initially, the vulnerability of the ten components is evaluated using an NN model with 126 neurons and trained accuracy of 96.5%. Figure 4 shows the Architectural Vulnerability Factor (AVF) of each component for the different SDC categories (Figure 2), in logarithmic scale. The AVF is proportionally weighted to the size of each component (3rd column of Table I). $COORD_{Ctrl}$ causes application crashes without any output in every single fault injection, thus, this component is excluded from all the presented results. We do not observe any crash for all the other components and, thus, for each of the remaining components we present the SDCs results. Figure 4 indicates that the component with the highest AVF for all the categories is the weight buffer (WB). In terms of critical errors (yellow bar), the WB is showing a considerable difference from all the other components. This happens because all the weights are stored in the WB from the beginning of an execution, therefore, a SEU will persist, manifesting a permanent fault. Hence, once a bit flip occurs in WB, it affects all of the remaining images.

The majority of the components (7 out of 9), express the same trend between the SDC categories with the exception of $ACT_{ctrl}$ that has more critical errors than all the other errors. A reasonable explanation lays in the activation function importance in the NN computation procedure. Applying a different activation function can drastically change the output of a neuron beyond the approximation nature of the algorithm causing mostly critical SDCs.

### C. Weight Buffer Performance - Reliability Trade-off Analysis

We further analyze the most critical component from the previous analysis (WB) by investigating how software refresh can help to reduce the SDC effects. In this way extra area overheads that common redundancy protection mechanisms require could be avoided. Thus, we analyse the idea of periodically overwriting the weights in the WB during the inference execution [24]. Table II shows the percentage of the reduced

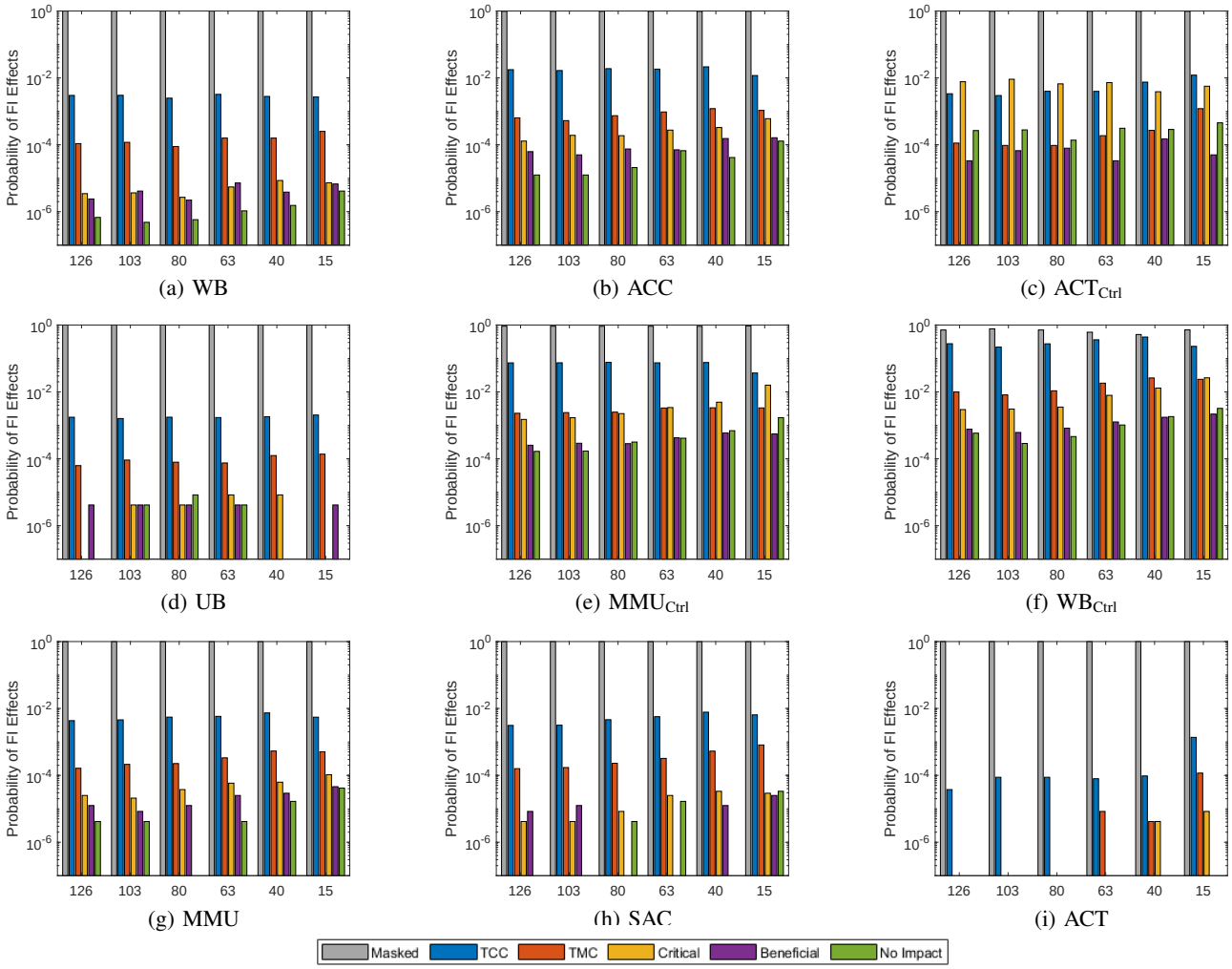


Fig. 5: Probability of a Fault Injection Effect for each model (ANN126, ANN103, ANN80, ANN63, ANN40, ANN15) for the tinyTPU's components (a) WB, (b) ACC, (c) ACT$_{Ctrl}$, (d) UB, (e) MMU$_{Ctrl}$, (f) WB$_{Ctrl}$, (g) MMU, (h) SAC, (i).

TABLE II: Performance Overhead and Percentage of SDC reduction in each Category for Different Refresh Rates Normalized with the No Refresh WB

| **Refresh Frequency (# Images)** | **Perf. Overhead (%)** | **TCC (%)** | **TMC (%)** | **Critical (%)** | **Benef. (%)** | **No Imp. (%)** |
|---|---|---|---|---|---|---|
| No Refresh | 0 | 0 | 0 | 0 | 0 | 0 |
| 5000 | 0.75 | 55.56 | 48.67 | 63.89 | 8.00 | 28.57 |
| 2500 | 2.23 | 72.70 | 65.34 | 61.11 | 64.00 | 74.43 |
| 1000 | 6.77 | 91.02 | 88.56 | 83.33 | 72.00 | 100 |
| 500 | 15.74 | 95.35 | 95.48 | 94.44 | 92.00 | 100 |

SDC errors for each category for different refresh rates, as well as the performance overhead when compared to a WB with no refresh. We consider four different refresh rates, by determining the number of the images processed between refreshes. As seen in Table II, when the refresh frequency increases, the different SDC errors for all the categories decrease. At the same time, the latency increases reaching up to 15.74% when the memory is refreshed every 500 processed images, where there is a significant reduction to all the SDC error categories. This experiment and analysis clearly motivates for the need of more sophisticated approaches to provide efficient and low overhead protection mechanisms for the WB in such accelerators.

### *D. Models Vulnerability Analysis at Component Granularity*

To investigate the effects of soft errors in ANN of different sizes (in terms of the number of neurons), we train different models ranging from 126 to 15 neurons in the hidden layer. The range was selected with the aim to reduce the NN area and at the same time allow only up to 5% accuracy degradation.

Table III shows the different ANN models with their respective accuracy and the number of cycles needed for an inference execution of 14 and 10000 images. This analysis allows the investigation of the trade-off between reliability and latency/area overhead for each component.

Figure 5 shows the probability of the fault injection effects to all tinyTPU components while inferencing all the ANN models listed in Table III. It is important to note that the presented results express the probability of fault injection

TABLE III: Neural Network Models

| Model Name | # of Neurons | Accuracy in % | MNIST cycles | |
|---|---|---|---|---|
| | | | 14 images | 10000 images |
| ANN126 | 126 | 96.5 | 7475 | 5337150 |
| ANN103 | 103 | 96.65 | 6747 | 4817358 |
| ANN80 | 80 | 96.24 | 5124 | 3658536 |
| ANN63 | 63 | 95.25 | 4316 | 3081624 |
| ANN40 | 40 | 94.4 | 2701 | 1928514 |
| ANN15 | 15 | 90.6 | 1892 | 1350888 |

(FI) effects after a fault is injected, hence the effect of the component's size is not incorporated in this analysis. In most of the components we can observe a general increasing trend in critical SDCs as the number of ANNs decreases. This happens due to the fact that smaller ANNs do not have a lot of redundancy and, thus, their neurons are more vulnerable to SEUs. The results for $ACT$ component, show that even though is a non-critical component with ANN126, when the number of neurons in the hidden layer of the ANNs decreases to 40 and 15, the critical errors appear and the component is turned from non-critical to a vulnerable component. Another observation is that the three controllers ($ACT_{ctrl}$, $MMU_{ctrl}$ and $WB_{ctrl}$) have the highest probability of manifesting a critical SDC as they provide the control signals and an error on those signals can affect the whole classification process.

The results clearly indicate that identifying the components to protect in a system for each NN model is not a straightforward process. Thus, such analysis needs to be done to guide selective protection decisions.

## VI. Conclusions

This paper provides a vulnerability analysis of an edge-based NN inference accelerator architecture by using a proposed evaluation methodology. The results presented in this work demonstrate that it is important to provide accurate and fine-grained vulnerability analysis, before making different design choices. The findings of this analysis point to several future directions, such as the investigation on how the combination of optimization techniques like pruning and quantization can affect the criticality of different components. Finally, it is important to investigate the generality of the observations made on the architecture used in this work, with alternative datasets, accelerator architectures and neural networks with more depth and different activation functions.

## References


[1] A. Mosenia, S. Sur-Kolay, A. Raghunathan, and N. K. Jha, "Wearable medical sensor-based system design: A survey," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 3, no. 2, pp. 124–138, 2017.

[2] C. Kyrkou, S. Timotheou, P. Kolios, T. Theocharides, and C. Panayiotou, "Drones: Augmenting our quality of life," *IEEE Potentials*, vol. 38, no. 1, pp. 30–36, 2019.

[3] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[4] Y. Ibrahim, H. Wang, J. Liu, J. Wei, L. Chen, P. Rech, K. Adam, and G. Guo, "Soft errors in dnn accelerators: A comprehensive review," *Microelectronics Reliability*, vol. 115, p. 113969, 2020.

[5] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[6] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[7] M. Goncalves, F. Fernandes, I. Lamb, P. Rech, and J. R. Azambuja, "Selective fault tolerance for register files of graphics processing units," *IEEE Transactions on Nuclear Science*, vol. 66, no. 7, pp. 1449–1456, 2019.

[8] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–14.

[9] A. Mahmoud, S. K. S. Hari, C. W. Fletcher, S. V. Adve, C. Sakr, N. Shanbhag, P. Molchanov, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Hardnn: Feature map vulnerability evaluation in cnns," *arXiv preprint arXiv:2002.09786*, 2020.

[10] A. Kouloumpris, T. Theocharides, and M. K. Michael, "Cost-effective time-redundancy based optimal task allocation for the edge-hub-cloud systems," in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2020, pp. 368–373.

[11] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech, "Demystifying soft error assessment strategies on arm cpus: Microarchitectural fault injection vs. neutron beam experiments," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 26–38.

[12] G. Li, K. Pattabiraman, and N. DeBardeleben, "Tensorfi: A configurable fault injector for tensorflow applications," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018, pp. 313–320.

[13] Y. He, P. Balaprakash, and Y. Li, "Fidelity: Efficient resilience analysis framework for deep learning accelerators," *MICRO*, 2020.

[14] A. Ruospo, A. Balaara, A. Bosio, and E. Sanchez, "A pipelined multi-level fault injector for deep neural networks," in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2020, pp. 1–6.

[15] A. Riefert, J. Müller, M. Sauer, W. Burgard, and B. Becker, "Identification of critical variables using an fpga-based fault injection framework," in *2013 IEEE 31st VLSI Test Symposium (VTS)*, April 2013, pp. 1–6.

[16] S. Di Carlo, P. Prinetto, D. Rolfo, and P. Trotta, "A fault injection methodology and infrastructure for fast single event upsets emulation on xilinx sram-based fpgas," in *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oct 2014, pp. 159–164.

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

[18] I. Chadjiminas, C. Kyrkou, T. Theocharides, M. K. Michael, and C. Ttofis, "In-field vulnerability analysis of hardware-accelerated computer vision applications," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2015, pp. 1–4.

[19] I. Chadjiminas, I. Savva, C. Kyrkou, M. K. Michael, and T. Theocharides, "Emulation-based hierarchical fault-injection framework for coarse-to-fine vulnerability analysis of hardware-accelerated approximate algorithms," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 830–833.

[20] "Etiny-tpu." https://github.com/jofrfu/tinyTPU, 2020.

[21] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks," *Communications of the ACM*, vol. 61, no. 9, pp. 50–59, 2018.

[22] "Yann Lecun, Corinna Cortes and Christopher J.C. Burges "The MNIST database of handwritten digits", http://yann.lecun.com/exdb/mnist/."

[23] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 502–506.

[24] A. Sari and M. Psarakis, "Scrubbing-aware placement for reliable fpga systems," *IEEE Transactions on Emerging Topics in Computing*, 2017.