Runtime Fault Injection Detection for FPGA-based DNN Execution Using Siamese Path Verification

Xianglong Feng, Mengmei Ye, Ke Xia, and Sheng Wei

Department of Electrical and Computer Engineering
Rutgers University, Piscataway, NJ, USA
Email: {xianglong.feng, mengmei.ye, ke.xia, sheng.wei}@rutgers.edu

2021 Design, Automation & Test in Europe Conference & Exhibition (DATE) | 978-3-9819263-5-4/21/$31.00 ©2021110.23919/DATE51398.2021.9473941

***Abstract*—Deep neural networks (DNNs) have been deployed on FPGAs to achieve improved performance, power efficiency, and design flexibility. However, the FPGA-based DNNs are vulnerable to fault injection attacks that aim to compromise the original functionality. The existing defense methods either duplicate the models and check the consistency of the results at runtime, or strengthen the robustness of the models by adding additional neurons. However, these existing methods could introduce huge overhead or require retraining the models. In this paper, we develop a runtime verification method, namely Siamese path verification (SPV), to detect fault injection attacks for FPGA-based DNN execution. By leveraging the computing features of the DNN and designing the weight parameters, SPV adds neurons to check the integrity of the model without impacting the original functionality and, therefore, model retraining is not required. We evaluate the proposed SPV approach on Xilinx Virtex-7 FPGA using the MNIST dataset. The evaluation results show that SPV achieves the security goal with low overhead.**

I. Introduction

Deep neural networks (DNNs) have been widely employed for a variety of artificial intelligence applications. Due to the heavy computation demands of DNNs, it has become a common practice to offload the DNN models to cloud platforms, which are equipped with rich computing resources and hardware accelerators (e.g., GPUs and FPGAs) [1]. In particular, FPGAs can achieve high performance, power efficiency, and design flexibility and thus become a powerful platform for DNNs [2].

However, FPGAs are also subject to a variety of security attacks and, given the importance of adopting FPGA for DNN acceleration, protecting the integrity of the FPGA-based DNNs have attracted researchers' attentions [3]. One of the vulnerabilities in FPGA is its sensitivity to fault injection attacks [4]–[7], which could flip certain bits in memory, and the flipped bits may alter the critical parameters of the DNN model and result in incorrect inference results.

There have been research works that target on defending against the fault injection attacks. For FPGAs, He et al. [8] check abnormal physical signals at runtime to detect injected faults. However, this approach only applies to specific devices and requires sophisticated sensors. D'Angelo et al. [9] employ the triple modular redundancy-based fault-tolerance technique to minimize the impact of injected faults, but the approach requires to triplicate the resource usage. To defend against fault injection attacks on DNNs, Li et al. [10] develop a dual modular redundancy framework (referred to as the "Two-Track" method in our discussion), which conducts runtime fault detection by introducing and comparing with a duplicated model. Libano et al. [7] and Clemente et al. [11] enhance the DNN robustness by adding a small set of neurons to the original model. These DNN defense methods typically require retraining the model.

In this paper, we aim to address the aforementioned limitations in the existing methods and develop an effective defense approach against fault injections on FPGA-based DNN models, with the following design principles and requirements. (1) *No retraining.* In the real-world settings, FPGA-based accelerators are typically deployed in the cloud server, and the client (e.g., a mobile device) has limited storage and computing resources. Therefore, after adding the verification structure in FPGA, the client should not be required to retrain or fine tune the DNN model. (2) *Runtime verification.* When the model and input data are uploaded to the cloud sever, the client loses control over how the cloud executes the model and produces the results. Therefore, it is important for the client to verify the correctness of the results at runtime. (3) *Easy deployment.* Different DNN implementations are dependent on different machine learning libraries. The verification method should be easily deployable to the original DNN model without causing library dependency or hardware compatibility issues. (4) *Controllable verification overhead.* With varying runtime constraints, the client should be able to dynamically select different strengths of verification and thus control the tradeoff between security and overhead.

We develop a Siamese Path Verification (SPV) method to check the integrity of the FPGA-based DNN at runtime. SPV leverages the computing features of the fully-connected layers in the DNN model to conduct runtime verification, and it does not rely on any specific devices or libraries. In SPV, new neurons are added to the trained model, and the output vector contains not only the original inference results but also the verification results that check the correctness of the inference results. It is worth noting that, by carefully designing the weights for the neurons in different layers, we ensure that the added neurons do not impact the original functionality of the DNN model and, therefore, SPV does not require model retraining. Also, we could control the verification overhead by configuring the number of neurons to construct SPV.

II. Threat Model

We target on the FPGA fault injection attacks that can compromise FPGA-based DNN execution, which could be

carried out by many methods, such as FPGAhammer [4], RAM-Jam [5], rowhammer [6], radiation [7], and power/clock glitches [12], with different characteristics in the causes and effectiveness of the injected faults. In our current work, we adopt an abstract and generic fault injection model, which does not differentiate the technical and physical differences of the various fault injection methods but considers the overall fault injection rate on DNNs (i.e., the percentage of the weights modified by the attack) as its main property for configuration. In the future work, we will extend the scope of our exploration from the generic/abstract model to the physical fault injection models that are specifically applicable to the remote cloud environment, such as FPGAhammer [4] and RAM-Jam [5]. The injected fault, such as flipped bits in sensitive weights or parameters, could be leveraged by the attacker to compromise the original functionality of the DNN model, resulting in wrong inference results [13], [14].

III. Verification with Siamese Path

We develop a new defense approach targeting on addressing the following two *Research Questions (RQs)*: (1) *RQ-1:* From the technical perspective, how could we add neurons to verify the integrity of the DNN at runtime based on existing machine learning libraries and hardware platforms? (2) *RQ-2:* From the perspective of the client, how could it verify the FPGA security without retraining the model and with flexible verification options to control the tradeoff between security and performance overhead? To address the two RQs, we employ a Siamese path verification (SPV) method for the verification of key weights. By carefully designing the weights, SPV leverages the existing features of the fully connected layers to retain the original functionality of the neural network and pass the verification results to the final output, which provides us with the advantage of runtime verification without retraining.

A. SPV Design

To address *RQ-1*, we construct the SPV by adding a set of neurons at different layers and weights to automatically check the integrity of the neural network and pass the verification results to the output along with the inference results. The detailed design of SPV is shown in Fig. 1, where the nodes represent the neurons and the lines represent the weights of the neural network. Specifically, the *green nodes* represent the target neurons for verification. The *red nodes* represent the intermediate probe nodes, which share the same weights (i.e., Siamese path) and are supposed to obtain the same values as that of the target neurons (i.e., the *green nodes*). The *orange nodes* are the probe nodes, which use a non-zero value to represent the case where fault injection is found in a Siamese path. The *blue lines* represent the target weights for verification, and the *red lines* are the copies of the target weights. The *green lines* appear in pairs, where one is "-1" and the other is "1" for one node. The *orange lines* are "1"s, which propagate the probe node's value to the final output layer.

To describe how SPV works in detail, we define that the *blue lines* (i.e., the original weight w) and the *red lines* (i.e., the new weight $\widetilde{w}$) are the same weights (i.e., $w = \widetilde{w}$) but connected

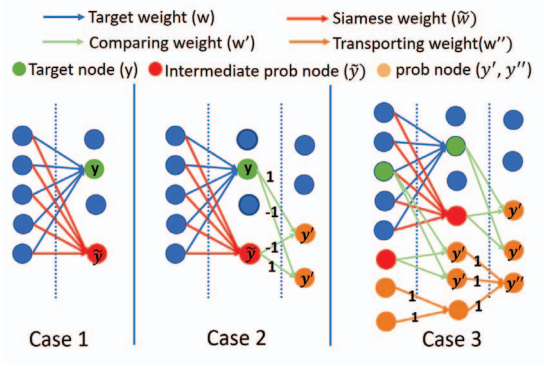

Fig. 1. The proposed Siamese path verification (SPV) design.

to two different nodes, where one is the original neuron (i.e., the *green node* y) in the neural network and the other is the intermediate probe node (i.e., the *red node* $\widetilde{y}$). We treat the two sets of weights as Siamese paths since they share exactly the same value (i.e., $w = \widetilde{w}$). Here we set the function of the fully connected layer as fc and the calculation with the input x is $y = fc(x)$. Given that $\widetilde{y} = fc(\widetilde{w})$ and $w = \widetilde{w}$, the values in the green node y should be equal to the red node $\widetilde{y}$.

The intermediate probe node is the base of the SPV node. Based on the layer where the nodes reside, there are three cases, as shown in Fig. 1. In different cases we need to design different weights and the probe nodes for passing the verification results to the output.

- **Case 1.** The target node (i.e., *green node*) and the intermediate probe node (i.e., *red node*) are in the next-to-last layer, which will be forwarded to the output layer via the Softmax layer. In this case, we just need to collect the output and compare the values of the *green node* and the *red node* as shown in Case 1 of Fig. 1. If the values are different, it indicates that the model is compromised by the fault injections.
- **Case 2.** The target node (i.e., *green node*) and the intermediate probe node (i.e., *red node*) are located prior to the next-to-last layer. In this case, we cannot directly compare the two values and thus we design a special weight w' along with the probe node y' in the next-to-last layer to compare the results and pass them to the output layer. The intuitive idea is to set the comparing weight as $w' = [-1, 1]$. Accordingly, the consecutive probe node is $y' = -1 \cdot y + 1 \cdot \widetilde{y}$, and we can tell that the weights are modified once $y' \neq 0$. However, the ReLU layer (i.e., $max(0, y')$) will set the negative values of y' to 0 and, as a result, we still obtain 0 when $y' < 0$. Here, we add one pair of probe nodes for one pair of Siamese paths (i.e., one *green node* and one *red node*) as shown in Case 2 of Fig. 1. More specifically, we set one probe node y'_1 with the weight $w'_1 = [-1, 1]$ and set the other pair of node y'_2 with the weight $w'_2 = [1, -1]$. Then, the two probe nodes are $y'_1 = -1 \cdot y + 1 \cdot \widetilde{y}$ and $y'_2 = 1 \cdot y - 1 \cdot \widetilde{y}$. Between y'_1 and y'_2, there is at least one node that is higher than 0 if the weight is modified. Therefore, the comparison result of the

target node and the intermediate probe node will not be lost after the ReLU layer.

- **Case 3.** When there are probe nodes in previous layers, we need to pass the values of those nodes to the output layer. In this case, all the probe nodes could be passed to one probe node in the consecutive layer as shown in Case 3 of Fig. 1. Given the n probe nodes (i.e., $y'_1 \dots y'_n$) in the previous layers, we set the corresponding new weights as all 1s (i.e., $w'' = [1, \dots, 1]$), namely transporting weights. The probe node in the consecutive layer is $y'' = \sum_{y'}^{n} y'_i$. Thus, the probe node is non-zero if there is fault injection detected.

Furthermore, we propose two strategies to minimize the resource usage and meet the *flexible verification overhead* requirement in *RQ-2*. The first strategy is to randomly select the nodes to design the Siamese path and, every time we upload the weights of the model, we use different sets of nodes that are randomly selected. In this way, we could verify more weights. The second strategy is that we analyze the weights first and then select the nodes that contain the largest number of weights with high absolute value to set the Siamese path for verification. In this paper, our implementation and evaluation are based on the second strategy.

B. Maintaining the Original Functionality

In this section, we will introduce the method of designing the remaining weights that are not covered in Section III-A. Fig. 2 shows the diagram of adding weights for new neurons without impacting the original functionality, where the matrix represents the weights, and the vector represents the neurons in one layer. More specifically, the blue matrix represents the original weight and the blue vector represents the original neurons. Given the input vector a, the first and second weight matrix A, A', the neurons in the third layer should be $a \bigotimes A \bigotimes A'$.

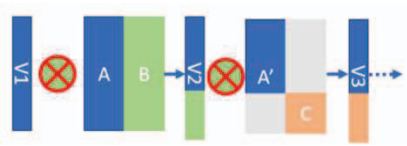

Fig. 2. Weight matrix and neuron vector in the DNN.

To add more neurons to one layer while meeting the *no retraining* requirement in *RQ-2*, we leverage the features of fully-connected layer to enlarge the weight matrix. Following Equation (1), we extend the original weight matrix A with the additional weight matrix B, written as $[AB]$. The new second layer with additional neurons is $[a \bigotimes A, a \bigotimes B]$, in which $a \bigotimes B$, represented as b, is the additional neurons (i.e., the intermediate probe nodes and the probe nodes).

$$a \bigotimes \begin{bmatrix} A & B \end{bmatrix} = \begin{bmatrix} a \bigotimes A, a \bigotimes B \end{bmatrix} \tag{1}$$

To further add neurons to the third layer, we enlarge the second weight matrix following Equation (2), where the new second weight matrix is $\begin{bmatrix} A' & \mathbf{0} \\ \mathbf{0} & C \end{bmatrix}$ with the original second weight matrix A', the additional weight matrix C and the zero padding matrix $\mathbf{0}$.

$$\begin{bmatrix} a \bigotimes A, b \end{bmatrix} \bigotimes \begin{bmatrix} A' & \mathbf{0} \\ \mathbf{0} & C \end{bmatrix} = \begin{bmatrix} a \bigotimes A \bigotimes A', b \bigotimes C \end{bmatrix} \tag{2}$$

The total neurons in the third layer contain two parts, namely the $a \bigotimes A \bigotimes A'$, which is the original neurons for the third layer, and the $b \bigotimes C$, which is the added neurons. For more layers, the rest of the propagation flow will follow Equation (2) to meet the *no retraining* requirement in *RQ-2*. In this way, we could add new neurons to the network without compromising its original functionality and, therefore, model retraining is no longer required.

IV. Experimental Results

A. Experimental Setup

In our system implementation and evaluation, we use a PC (Intel i7, 16GB) with Xilinx Virtex-7 acceleration card as the server. We adopt a DNN for handwriting recognition as the target DNN and implement it on the FPGA using Xilinx SDAccel 2018.2. Fig. 3 shows the structure of the DNN model. The input is a gray-scale image, which will be pre-processed and reshaped into a vector with the length of 784. The neural network consists of three layers with 200, 50 and 10 neurons. The 10 nodes in the last layer represent the 10 digits from 0 to 9. There is one ReLU layer between each pair of consecutive layers. Based on this DNN model, we evaluate the impact of fault injection on the recognition accuracy with and without SPV applied using the MNIST dataset [15]. Also, we evaluate the overhead of SPV, in comparison with the Two-Track method [10], in terms of the model size increase and the model execution time.

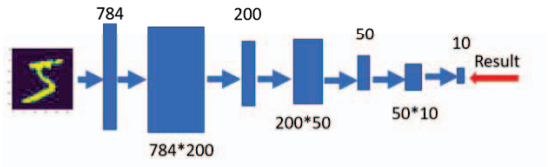

Fig. 3. Structure of the DNN adopted in the evaluation of SPV.

B. Fault Injection Attack

We generate the fault injection samples in the DNN model by following the method introduced in [16]. Also, based on the work from [13], [14], we notice that some key weights are sensitive to the fault injection attacks. Therefore, in our evaluation, we assume that the attacker will conduct fault injection to the key weights (i.e., those that contain high absolute values). The results of the fault injection attacks are shown in Fig. 4(a), with the consideration that the top 1% to 9% of the weights (based on their absolute values) are modified by the injected fault. We observe that, without the protection of SPV, the accuracy could be reduced to lower than 50% by modifying only 7% of the weights, indicating the effectiveness of the fault injection attack on the target DNN model.

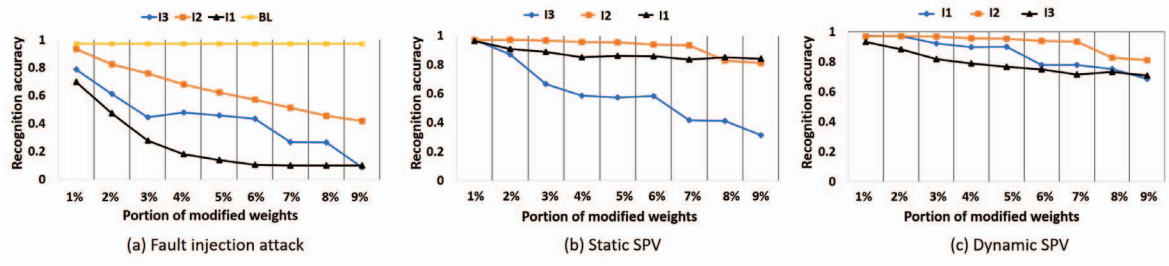

Fig. 4. Handwriting recognition accuracy with fault injection. I1, I2, and I3 represent the model with fault injection on high absolute weights of layers 1, 2 and 3, respectively. "BL" represents the original model. Y-axis represents the recognition accuracy, and X-axis represents the portion of the modified weights.

C. SPV-based Defense

We evaluate the performance of SPV by assuming that it protects $P\%$ of the model weights, with the following test cases. (1) *Static SPV*, where we set a static P value for each layer (i.e., $P = 30$). The results are shown in Fig. 4 (b), where we observe that layer 3 is the most sensitive to the fault injection attack and layer 1 is the least sensitive. (2) *Dynamic SPV*, where we set a dynamic P value for each layer and, in particular, $P = 20$, 20, and 60 for layers 1 to 3 in our experiments. Fig. 4 (c) shows the recognition accuracy results when Dynamic SPV is applied, where the overall recognition accuracy is significantly higher than the Static SPV method.

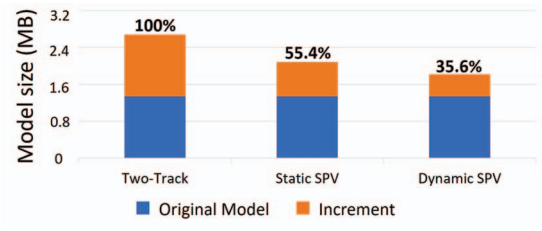

Fig. 5. Comparison of model size increase caused by Two-Track, Static SPV and Dynamic SPV.

Fig. 5 compares the size of the model when using the Two-Track method, Static SPV and Dynamic SPV. The blue bar shows the original size of the model, and the orange bar shows the size increase after applying the defense mechanisms. We observe that Dynamic SPV introduces significantly lower overhead in model size than Two-Track and Static SPV. Furthermore, we evaluate the total execution time of the original model and the three defense methods for handwriting recognition of 10,000 test images from the MNIST dataset, which are shown in Fig. 6. We observe that Dynamic SPV achieves more than 60% of timing reduction compared to the Two-Track method.

V. Conclusion

We developed a runtime fault injection detection method for DNNs on FPGAs, namely SPV, by designing a Siamese path verification framework. SPV inserts verification neurons into the DNN model without compromising the original functionality or retraining the original model. The evaluation results showed that SPV achieves high effectiveness and efficiency defending against fault injection attacks on FPGA-based DNN.

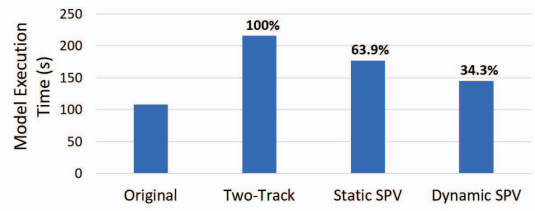

Fig. 6. Comparison of model execution time (seconds) for handwriting recognition of 1000 test images with the original neural network, Two-Track, Static SPV and Dynamic SPV.

Acknowledgement

We appreciate the constructive reviews provided by the anonymous reviewers. This work was supported in part by the National Science Foundation under Award CNS-1912593.

References

[1] "Amazon EC2 F1 instances," aws.amazon.com/ec2/instance-types/f1/.
[2] Y. Guan *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *FCCM*, 2017, pp. 152–159.
[3] X. Xu *et al.*, "Rethinking FPGA security in the new era of artificial intelligence," in *ISQED*, 2020, pp. 46–51.
[4] J. Krautter *et al.*, "FPGAhammer: Remote voltage fault attacks on shared FPGAs, suitable for DFA on AES," *TCHES*, pp. 44–68, 2018.
[5] M. M. Alam *et al.*, "RAM-Jam: Remote temperature and voltage fault attack on FPGAs using memory collisions," in *FDTC*, 2019, pp. 48–55.
[6] Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA*, 2014, pp. 361–372.
[7] F. Libano *et al.*, "Selective hardening for neural networks in FPGAs," *IEEE Transactions on Nuclear Science*, vol. 66, no. 1, pp. 216–222, 2018.
[8] W. He *et al.*, "An FPGA-compatible PLL-based sensor against fault injection attack," in *ASP-DAC*, 2017, pp. 39–40.
[9] S. D'Angelo *et al.*, "Fault-tolerant voting mechanism and recovery scheme for TMR FPGA-based systems," in *DFT*, 1998, pp. 233–240.
[10] Y. Li *et al.*, "D2NN: a fine-grained dual modular redundancy framework for deep neural networks," in *ACSAC*, 2019, pp. 138–147.
[11] J. A. Clemente *et al.*, "Hardware implementation of a fault-tolerant hopfield neural network on FPGAs," *Neurocomputing*, vol. 171, pp. 1606–1609, 2016.
[12] L. Zussa *et al.*, "Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism," in *IOLTS*, 2013, pp. 110–115.
[13] P. Zhao *et al.*, "Fault sneaking attack: A stealthy framework for misleading deep neural networks," in *DAC*, 2019, pp. 1–6.
[14] Y. Liu *et al.*, "Fault injection attack on deep neural network," in *ICCAD*, 2017, pp. 131–138.
[15] "MNIST database," 2020, http://yann.lecun.com/exdb/mnist/.
[16] J. Tonfat *et al.*, "Method to analyze the susceptibility of HLS designs in SRAM-based FPGAs under soft errors," in *ARC*, 2016, pp. 132–143.