Evaluating Data Encryption Effects on the Resilience of an Artificial Neural Network

Riccardo Cantoro[1], Nikolaos I. Deligiannis[1], Matteo Sonza Reorda[1], Marcello Traiola[2], Emanuele Valea[3]
[1]*Politecnico di Torino, Dip. Automatica e Informatica*, Torino, Italy
{riccardo.cantoro|nikolaos.deligiannis|matteo.sonzareorda}@polito.it
[2]*INL, École Centrale de Lyon*, Lyon, France – marcello.traiola@ec-lyon.fr
[3]*LIRMM, Université de Montpellier / CNRS* , Montpellier, France – valea@lirmm.fr

***Abstract*—Nowadays, many electronic systems store valuable Intellectual Property (IP) information inside Non-Volatile Memories (NVMs). Therefore, encryption mechanisms are widely used in order to protect such information from being stolen or modified by human attacks. Encryption techniques can be used for protecting the application code, or sensitive sets of data in the NVM. In particular, in machine-learning applications, the weights of an Artificial Neural Network (ANN) represent a highly valuable IP stemming from long time invested in training the system along the development phase. On the other side, systems implementing ANN applications are increasingly used in safety-critical domains (e.g., autonomous driving), where a high reliability level is required. In a previous paper, we have shown that encryption techniques, applied to the application code of generic systems, provide a significantly higher error detection rate. In this paper, we focus on an ANN application and we evaluate the detection rate induced by encryption mechanisms for transient faults possibly impacting the ANN weights. We performed experiments on a pre-trained ANN, whose weights represent the sensitive IP of our system. We executed fault injection campaigns to evaluate the ANN resilience when different encryption methods are used. Experimental results showed that the presence of specific encryption mechanisms alone induces high fault detection rates in such applications. This may allow the designer to consider security and safety mechanisms together, achieving the same results with lower costs.**

I. Introduction

In the last decades, digital systems have become widespread, considerably changing the way people interact with computing machines. This strong development has imposed design constraints that were not crucial when computing systems did not manage aspects related to the safety and security of humans' life. For this reason, design guidelines for safety-critical applications aim to obtain *reliable* systems that are able to detect a sufficiently high percentage of faults that could affect the correct operation of the target system. Machine learning is a prominent example in this direction. In the last years, several safety-critical domains have been empowered by machine-learning, such as autonomous driving, robotics and health [1].

Furthermore, malicious users can represent a threat for the integrity of the system, in the case in which they have physical access to the device and they are able to perform attacks to extract valuable secret information from the device. For this reason, *memory encryption* techniques are adopted in order to protect data stored into memories.

Artificial Neural Networks (ANNs) for autonomous driving are an example of an application where both reliability and security are essential requirements. On the one hand, hardware security for ANNs is focused on the protection of training data that are stored into non-volatile memories (NVMs). Indeed, training data derive from a long and expensive process, whose results represent a valuable intellectual property for the system designer [2], [3]. On the other hand, faults affecting the NVMs have to be detected and possibly tolerated to avoid catastrophic consequences. Redundancy solutions are commonly applied in the automotive domain to detect the occurrence of single- and multiple-bit errors and to possibly correct some of them. For instance, Error Correction Codes (ECCs) are commonly used to detect (and possibly correct) errors up to a given multiplicity [4]. ECCs resort to redundancy (some extra bits are generated and stored) to check for data integrity. This involves both hardware and performance overhead. In the literature, alternative solutions have been explored. For example, the work from Martínez et al. [5] proposes techniques to increase the probability that the processor triggers an exception when a faulty instruction is fetched. However, this solution requires the modification of the CPU core design, which is often not possible for system companies using off-the-shelf processors for their applications.

In this paper, we explore the transversal characteristic of memory encryption, which may augment data corruption in the presence of faults affecting the system. Such an effect opens some opportunities for fault tolerance mechanisms. In a previous study [6], we showed that augmenting the corruption of an application code leads to increasing the fault detection probability for single faults and especially for multiple faults. This interesting property can be useful to optimize the utilization of costly ad-hoc mechanisms to detect faulty conditions. In this work, we investigate the fault detection capabilities of memory encryption mechanisms when applied to data. This study is important from a system designer perspective, who can thus take informed decisions about the choice of the system components.

In this work, we consider an ANN as a case study, whose weights are encrypted with different encryption mechanisms. We perform fault injection campaigns on the encrypted data, and we evaluate the impact of the encryption mechanism on the fault detection capability. Experimental results show that, by performing a simple padding check after the decryption, important fault detection capabilities can be achieved. This work paves the way to cleverly select the encryption mechanism not only to suitably protect the memory content with respect to malicious attacks, but also to achieve a sufficiently high degree of reliability.

The remainder of the paper is organized as follows. Section II briefly describes the background of the work. Section III introduces the reference scenario and details the proposed contribution

978-1-7281-9457-8/20/$31.00 ©2020 IEEE

of this work. Section IV illustrates the case study and Section V the obtained experimental results. Finally, Section VI draws the conclusions.

II. Preliminaries and Specifications

We previously evaluated the error detection capabilities that specific encryption mechanisms can provide when applied to the application code, stored in NVMs [6]. In that context, a fault may turn an instruction into an illegal one, causing a failure in the code execution. We showed that block ciphers amplify the fault effect, propagating it to a large number of bits. This raises the likelihood of an illegal instruction exception to be triggered, thereby significantly enhancing the system's fault detection capabilities thus complementing ad-hoc fault detection mechanisms. Specifically, single bit-flip fault detection values provided by the encryption/decryption mechanism alone were found to span from 81.5% to 93.5%. Moreover, block-ciphers proved to own interesting multiple-bit fault detection properties.

In this paper, we focus on how sensitive data, that is also stored in NVMs, can be protected against faults by only resorting to the encryption/decryption mechanisms. We consider here a pre-trained ANN implementing a classifier. In this case, the sensitive information is represented by the weights of the network. We are particularly interested in the weights of the classifier for two main reasons: (i) the weights, which are the outcome of the training process, play a significant role since they determine the overall classification accuracy of the network. They are typically considered as a precious Intellectual Property (IP). (ii) A fault affecting the ANN weights is particularly hard to detect without dedicated mechanisms, unless it triggers specific conditions handled by the software. The next subsections detail the background concepts and mechanisms used in this study.

A. Memory Encryption

The most common techniques for the encryption of data stored into memories are based on symmetric cryptography. In this scenario, the software developer loads the encrypted data inside an NVM that is usually external to the target System-on-Chip (SoC). The encryption is performed using a secret key, which is also stored inside the SoC, in a tamper-proof internal memory. When the device is running, all data fetched from the NVM are decrypted on-the-fly by a dedicated hardware decryption module, which generates the plaintext data that is processed by the CPU. These security techniques aim at protecting the target device against attackers that could tamper with the external NVM, stealing data with the purpose of cloning the system. In an ANN scenario, the network weights are a valuable IP, which could be the target of attacks aiming at the replication of the machine-learning model.

The encryption and decryption operations are carried out resorting to cryptographic functions that are based on pseudo-random permutations, such as the Advanced Encryption Standard (AES). The AES function takes as input a block of 128 bits and the secret key, and it outputs another block of 128 bits whose value is an encrypted version of the input. Therefore, it is not possible to correlate the output and the input in any way without knowing the secret key. The AES can be used to build *block ciphers* and *stream ciphers*.

Block ciphers take as input the message to cipher (i.e., the plaintext) one block at a time, and they output the result (i.e., the ciphertext) in blocks of the same size. In the case of the decryption function, the inverse operation is performed on the ciphertext to obtain the plaintext. The AES function can be used to create different kinds of block ciphers, according to the adopted configuration: (i) Cipher Block Chaining (CBC) mode; (ii) Cipher Feedback (CFB) mode; (iii) Propagating CBC (PCBC) mode.

Stream ciphers encrypt and decrypt data one bit at a time. The AES function can be used to build stream ciphers as well. In this case, the AES generates a stream of random bits, starting from a seed value. The generated stream (called *keystream*) is combined with the plaintext one bit at a time with a XOR operation. In the decryption process, the same operation is performed between the ciphertext and the keystream. The AES function can be used to create different kinds of stream ciphers, according to the configuration used to create the keystream: (i) Counter (CTR) mode; (ii) Output Feedback (OFB) mode.

Encryption schemes based on block-based functions, such as AES, show some implementation difficulties when the size of the data to encrypt/decrypt is not multiple of the block size. In this case, a padding of the plaintext is necessary. Several protocols exist in order to synchronize the padding. One of the most popular is the PKCS #7 - RFC 2315 [7]. According to this standard, the last encryption block must be completed with N bytes encoding the value 'N'. As we will show later on, the padding system can represent an interesting opportunity for the purpose of error detection.

For the purpose of this paper, we focus on the decryption function, which is the only function executed inside the device (the encryption function is performed off-line by the programmer). Moreover, we focus on a specific property of the decryption function, i.e., the peculiarity of propagating a 1-bit corruption of the ciphertext to multiple bits of the resulting plaintext.

B. Fault Model and Fault Classification

In order to model the effects of possible transient faults affecting the NVM, in this paper, we consider the *Single Event Upset (SEU)* and the *Multiple Bit Upset (MBU)* fault models. We perform fault injection campaigns only on the ANN weights. Each fault is classified according to its effects on the ANN application as: • *Silent*: the fault does not affect the execution of the program nor the results. • *Silent Data Corruption* (SDC): the fault does not affect the execution of the program but it affects the results. • *Detected*: (i) the fault affects the program execution by turning a valid floating point number into a NaN value and thus causes a *software exception*; (ii) the fault corrupts the padding bytes of the ciphertext and a padding check action detects the anomaly. Obviously, the desirable conditions are that a fault is either *silent* or *detected*. Indeed, we want to avoid scenarios where anomalous conditions occur and corrupt the system without warning, i.e., SDC. As already extensively studied [8]–[10], ANN-based systems are intrinsically rather resilient to errors. A fault affecting at the LSBs of an ANN weight value will probably have no effects on the execution of the application, thus it would be a silent fault. As the fault location moves towards the MSBs, its effects would be more severe and can lead to silent data corruption during execution [11], [12]. Faults located in the exponent segment of the floating point number could lead to NaN values generation. In the case of a NaN, the software is responsible of detecting it and handling it by raising an exception that will lead to the detection of the fault.

III. Reference Scenario

A. No Encryption Scenario

When no encryption is used to protect the NVM that stores the network's weights, a fault in the memory content will result in the

flipping of a bit (bit-flip). The criticality of the fault strongly depends on the fault location as explained in Subsection II-B. As mentioned in Subsection II-B, in this case the only way to detect a fault is that it causes a NaN value, which in turns triggers an exception.

B. Encryption Scenario

In this scenario, the encryption is used to protect the NVM content. As reported in Subsection II-A, a requirement for the data to be correctly encrypted is to have a size multiple of the block size. Thus, data are often previously padded before the encryption to satisfy such requirement. This means that some extra bytes are appended to the data before encryption, and they are removed after decryption. Before the removal, the integrity of the padding segment can be checked or not. Depending on that, we differentiate two possible sub-scenarios: 1) *Padding integrity check:* the first scenario concerns the system's software or hardware being able to perform checks on the padding bytes after the decryption. 2) *No padding integrity check:* in the second scenario the system does not perform any kind of check on the padding bytes when decrypting.

When a fault corrupts the encrypted memory content, its effect may propagate to the padding, thanks to the decryption. In this case, the padding check mechanism immediately detects that the padding bytes have been altered, thus leading to a detection of the fault.

On the other hand, if the system does not perform the padding integrity check after decryption, then it is up to the application to possibly detect the fault. As already mentioned, in our scenario this happens only if a NaN value is generated.

IV. Case Study

The ANN that we used for our experiments is a classifier, which was developed using an ANSI C library [13]. It is responsible, given a point in the (x, y) plane, to classify it in one of the three classes:

- C1: The point belongs to either one of the circles: $(x \pm 1)^2 + (y \pm 1)^2 \geq 0.16$
- C2: The point belongs to either one of the disks: $0.16 < (x \pm 1)^2 + (y \pm 1)^2 < 0.64$
- C3: The point belongs to neither circle nor disk: $(x \pm 1)^2 + (y \pm 1)^2 \geq 0.64$

The training and the testing set of the network contain $3,000$ points each ($6,000$ in total). In each set $1,500$ randomly generated points are located inside the $[0, 2] \times [0, 2]$ rectangle and $1,500$ points are located inside the $[0, -2] \times [0, -2]$ rectangle.

The network is composed of 1 input layer, 3 hidden layers and 1 output layer. The input layer has 2 neurons, one for each of the points coordinates (x, y). The hidden layers have 10 neurons each and the output layer has 3 neurons, one for each of the classes (C1, C2, C3). In total, the network contains 283 weights (including each neuron's BIAS input weight). In order to train the network we used the supervised learning technique. Every point in our training and test dataset was encoded using one-hot encoding. The adopted training algorithm was *gradient descent*.

The generalization error of the network was found to be 1.33%. This is the probability for the classifier to misclassify a given point of the test-set (e.g., to classify a point of the class C1 as a point of the class C2 or C3). The activation function selected for this network is the *sigmoid function*.

V. Results

The results of the experiments that consider SEU as the fault model are reported in Table I, while the results of the experiments that consider MBU as the fault model are plotted in the graphs in Figures 1 and 2. In particular, since we are interested in evaluating the resilience of the ANN, in the graphs we report the percentage of cases where a fault occurrence is considered to be 'safe', i.e., silent and/or detected (resilience rate).

A. NO PAD: Experiments Without Padding Checks

Regarding the SEU experiments, in the left part of Table I we can see that there is no big difference between the results of the experiments performed with no encryption and those with a stream-cipher-based encryption. As for the block ciphers, in this case they do not improve the detection rate and also tend to produce more SDCs than the other scenarios.

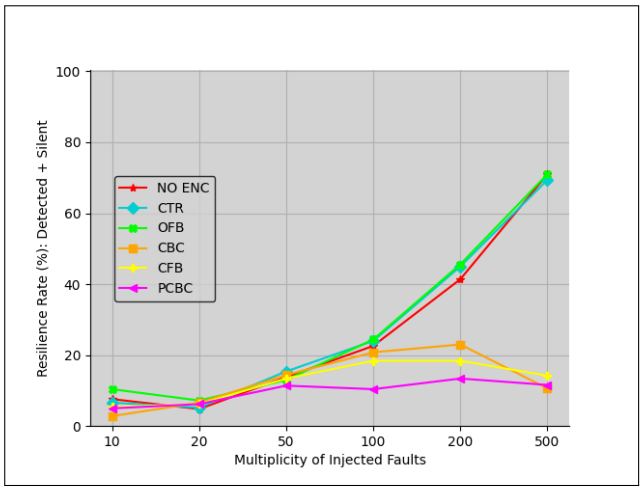

Fig. 1. MBU results: NO PAD experiments

By analyzing the MBU experiments' results in Figure 1, we can see that, with respect to the NO ENC experiments, we have a very small improvement of ANN resiliency in the case of stream ciphers. Block ciphers, on the other hand, seem to under-perform, since their detection rates are lower than those of the NO ENC case. Specifically, for fault multiplicity higher than 200 the block ciphers' detection rates drop. These results show a contrasting trend compared to our previous work [6], where we found that the block ciphers have a high detection capability for faults occurring in the code memory space (see Section II). Indeed, a fault may transform an instruction into an illegal one, causing a failure in the code execution. When block ciphers are used to encrypt the code, they amplify the fault effect, propagating it to other bits. This, in turns, raises the likelihood of an illegal instruction.

On the other hand, since in this work we focus on application data, the only possible fault detection scenario is for the fault to generate a NaN value which is detected by the software. In our experiments, stream ciphers were found to be able to generate NaN values more frequently than block ciphers.

The next subsection shows that, by simply checking the padding after the decryption, important results can be achieved.

B. PAD: Experiments With Padding Checks

In the right part of Table I we report the results in case of SEUs for PAD experiments. In these experiments, we evaluate the ANN resilience when padding byte checks are performed after the decryption. The ANN resilience when stream ciphers are used remains substantially unchanged. Unfortunately, CBC and CFB block

TABLE I
SEU RESULTS

Fault Class		Encryption									
		NO PAD					PAD				
		Stream ciphers		*Block Ciphers*			*Stream ciphers*		*Block Ciphers*		
	NO ENC	CTR	OFB	CBC	CFB	PCBC	CTR	OFB	CBC	CFB	PCBC
Silent	75.5%	78.8%	77.4%	1.6%	2.6%	6.4%	76.6%	76.6%	0.6%	1.8%	0%
SDC	24%	21.8%	22%	98%	97.4%	92.8%	22.8%	22%	98.2%	96.4	0.6%
Detected	0.3%	0%	0.6%	0.4%	0%	0.8%	0.6%	1.4%	1.2%	1.8%	99.4%

NO PAD: Experiments that do not consider padding checking while decrypting
PAD: Experiments that consider padding checking while decrypting

ciphers still provide an important amount of SDCs. Nonetheless, it is remarkable that the PCBC block cipher configuration is able to achieve a fault detection percentage of 99.4%.

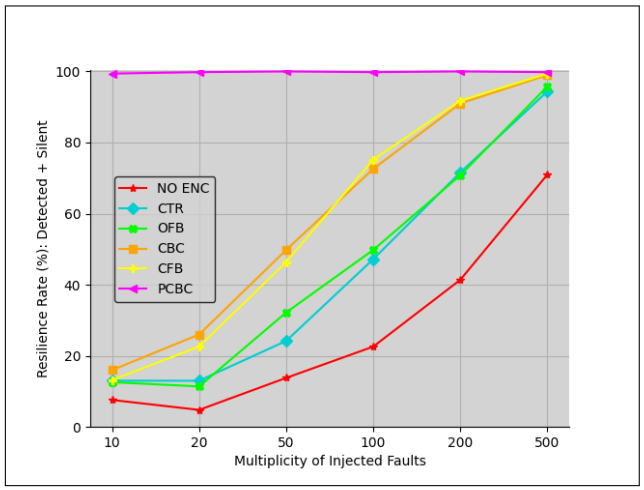

Fig. 2. MBU results: PAD experiments

As shown in Figure 2, a significant improvement is observed also in the MBU experiments. Both stream and block ciphers improve the ANN resiliency more than the NO ENC scenario. Block ciphers perform better than the stream ciphers. In this scenario, we obtained a trend similar to the one observed in our previous study [6]. Finally, regardless of the multiplicity of the injected faults, the PCBC configuration achieves $\approx$ 100% detection rate.

In conclusion, by using the AES PCBC configuration with pad checking to protect the IP stored in an NVM, it is possible to achieve a fault detection rate greater than 99%, even without resorting to costly ad-hoc fault detection mechanisms. Moreover, the padding check allows achieving an early detection, compared to spending time in executing the software and catching an exception due to possible NaN values.

By using this information, a system designer who needs to adopt encryption mechanisms can take informed decisions about the most suitable configuration, taking into account both the reliability and the security requirements.

VI. CONCLUSIONS

Artificial Neural Networks represent a good example where both hardware security and reliability requirements are crucial. In particular, ANN weights are often considered as Intellectual Property, thus they need to be protected from malicious attacks. Moreover, they play a significant role in determining the overall ANN classification accuracy, thus they must be protected against possible hardware faults. In this article, we studied the opportunities offered by the memory encryption as fault detection mechanism. The main goal is to help designers to take informed decisions when performing design choices. We performed fault injection campaigns on encrypted ANN weights. Experimental results highlighted that it is possible to achieve important fault detection rates ($> 99\%$) by using a particular AES configuration, i.e., PCBC with padding check. We are now extending our analysis to other application domains (e.g., in the automation area) where the effects of data and code encryption may result in similar reliability enhancements.

REFERENCES

[1] Y. LeCun *et al.*, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[2] F. Tramèr *et al.*, "Stealing machine learning models via prediction apis," in *25th USENIX Conference on Security Symposium*, ser. SEC'16. USA: USENIX Association, 2016, p. 601–618.

[3] S. Katzenbeisser *et al.*, "Security in Autonomous Systems," in *2019 IEEE European Test Symposium (ETS)*, 2019, pp. 1–8.

[4] J. Xiao-bo *et al.*, "Novel ECC structure and evaluation method for NAND flash memory," *IEEE International System-on-Chip Conference*, 2015.

[5] J. A. Martínez *et al.*, "A Scheme to Improve the Intrinsic Error Detection of the Instruction Set Architecture," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 103–106, July 2017.

[6] R. Cantoro *et al.*, "Evaluating the Code Encryption Effects on Memory Fault Resilience," in *2020 IEEE Latin American Test Symposium (LATS)*.

[7] B. Kaliski, "RFC2315: PKCS #7: Cryptographic Message Syntax Version 1.5," USA, 1998.

[8] W. Sung *et al.*, "Resiliency of Deep Neural Networks under Quantization," *CoRR*, vol. abs/1511.06488, 2015. [Online]. Available: http://arxiv.org/abs/1511.06488

[9] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Computing and Applications*, Oct 2018. [Online]. Available: https://doi.org/10.1007/s00521-018-3761-1

[10] C. Torres-Huitzil and B. Girau, "Fault and Error Tolerance in Neural Networks: A Review," *IEEE Access*, vol. 5, pp. 17 322–17 341, 2017.

[11] G. Li *et al.*, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126964

[12] A. Bosio *et al.*, "A Reliability Analysis of a Deep Neural Network," in *2019 IEEE Latin American Test Symposium (LATS)*, March 2019, pp. 1–6.

[13] Lewis Van Winkle, "C Neural Network Library: Genann," https://codeplea.com/genann, [Online; accessed 24-April-2020].