RNN-based Detection of Fault Attacks on RSA

Köylü, Troya Cagil; Reinbrecht, Cezar Rodolfo Wedig; Hamdioui, Said; Taouil, Mottaqiallah

DOI
10.1109/ISCAS45731.2020.9180708

Publication date
2020

Document Version
Accepted author manuscript

Published in
2020 IEEE International Symposium on Circuits and Systems (ISCAS)

RNN-Based Detection of Fault Attacks on RSA

Troya Çağıl Köylü, Cezar Rodolfo Wedig Reinbrecht, Said Hamdioui, Mottaqiallah Taouil
Computer Engineering, Delft University of Technology
Delft, the Netherlands

***Abstract*—Physical fault injection attacks are becoming an important threat to computer systems, as fault injection equipment becomes more and more accessible. In this work, we propose a new strategy to detect fault attacks in cryptosystems. We use a recurrent neural network (RNN) to detect problems in the program flow caused by injected faults. Our neural network is trained using the instructions of non-faulty operations and therefore, it can protect against both current and future attacks. As a case study, we use two implementations of software RSA. To test the effectiveness of our detector, we propose a collection of fault injection models, where each model represents different types of faults in the instructions. Evaluation results show that we obtain a high detection accuracy in case injected faults lead to changes in the instruction flow and hence, making it difficult to steal secrete keys. Finally, we propose an efficient hardware implementation with only a 6% area overhead compared to a RISC-V processor.**

I. Introduction

Faults affect the integrity of hardware and hence their impact should be investigated [1]. In the past, a lot of research has been conducted against faults caused by radiation, such as single upset events [2], [3]. Nowadays, attackers have the ability to create such faults artificially in a controlled environment [4], [5]. The target of such attacks is to disrupt the functionality or steal secret data such as keys. Simple techniques such as voltage underfeeding [6] and heating [7] are occasionally enough to achieve these malicious goals. If not successful, more complex means can be used such as EM waves or lasers [8]. Moreover, many researches have identified weaknesses in the implementation of cryptographic algorithms (such as RSA, which is the focus of this paper) and demonstrated that when the execution is glitched at the right moments, secure keys could be retrieved. Hence, it is important to protect sensitive algorithms such as crypto functions against fault injection attacks.

Two types of countermeasures reported in the literature can be used to protect RSA against fault injection: *prevention* and *detection*. In *prevention*, the countermeasures try to prevent the fault injection from occurring in the first place. In this category only passive shielding has been proposed. Passive shielding covers parts of the circuit in order to make it hard for electromagnetism or light pass through and hence hard to create faults [1]. Note that such countermeasures are very limited. The far majority of countermeasures are based on *detection*. We can further categorize *detection* into three subgroups. The first subgroup uses active shields, which continuously checks the integrity of data against an EM or laser attack [1], [9]. The second subgroup uses sensors to detect fluctuations in light, supply voltage, and clock frequency as a result of glitching [1]. However, both active shields and sensors work only for specific fault attacks. The third subgroup uses integrity checks of the sensitive operation by adding redundancy. They are the most popular countermeasures. Redundancy can be added over time [10], [11] or in space [12], [13]. In RSA, redundancy is attained by calculating the whole algorithm or parts of it again [14], calculating the inverse operation [15], or using a validation operation [16]–[18]. Although the detection mechanisms based on redundancy work well against all fault injection techniques, they can be bypassed when the final redundancy check is glitched. In general, all of the above countermeasures clearly have limitations and a protection scheme that addresses them is needed.

In this paper, we develop a novel and *intelligent* detection countermeasure based on a recurrent neural network (RNN). The RNN evaluates the integrity of the program flow of the RSA decryption. Major benefits of this approach are i) generality - it works against all fault injection techniques that affect the program flow; ii) reconfigurability - the neural network can be modified for different crypto algorithm implementations by modifying its weights; and iii) robustness - there is no final check, as each instruction is verified individually, and this makes bypassing the fault injection check extremely hard. As a result, the main contributions of this paper are the following:

- Proposal and development of a generic, reconfigurable and robust RNN-based detector against fault injection attacks on software RSA.
- Development of a validation methodology to evaluate the effectiveness of the proposed detector against real threats using custom fault injection models.
- Proposal of an efficient and optimized hardware architecture of the detector with an acceptable overhead.

The remainder part of this paper is organized as follows. Section II explains the scope of the work. Section III presents the design of the detector. Section IV describes the experimental setup and results. Finally, Section V discusses the results and reflects on the proposed detector.

II. Methodology

This section describes the design and evaluation of our RNN-based detector. The methodology consists of six steps, which are described further.

1. Selection of the target application: In general, the detector can be trained to learn any instruction sequence

and hence, can be applied to any secure application. However, in this paper we focus on the protection of software RSA [19] decryption implementations based on square-and-multiply (SAM) and Chinese remainder theorem (CRT) [20].

2. Selection of the instruction set architecture (ISA): To execute an RSA software implementation, its code must be compiled into machine instructions. These machine instructions depend on the selected ISA. We use a RISC-V architecture [21] in this work.

3. Definition of the threat model: During an RSA execution, instructions are vulnerable to attacks. When the right instructions are glitched, RSA can be broken. We assume that an attacker has access to the outputs of correct and faulty decryptions. Hence, the attacker is able to exploit threats referred as Bellcore [22] and Bao [23].

4. Creation of the fault injection models: To realize the threats, faults have to be injected. We consider the following four fault injection models:

1) *Bit-level fault model*. This fault model injects a single bit flip in an instruction. The position of the bit is selected randomly. This fault model reflects the effects of high-end fault injection techniques. For example, (Agoyan *et al.*, 2010) used a laser to flip single bits [8].
2) *Byte-level fault model*. This fault model selects a random byte and modifies it randomly to another. This fault model reflects the effects of a simpler fault injection technique. For example, (Barenghi *et al.*, 2011) used voltage underfeeding to create byte faults [24].
3) *Branch-to-opposite fault model*. This fault changes branch instructions to the opposite. We created this fault model to address the practical attack presented by (Barenghi *et al.*, 2009). In this attack, the authors used this strategy with voltage underfeeding, which enabled them to realize Bao's threat [25].
4) *Instruction-to-instruction-I/-II fault models*. These two fault models are extensions of the previous one. They randomly change one instruction into another. In variant I, a limitation is set for the branch instruction type; only branch instructions can change into other branch instructions. This is to limit the number of crashes. This limitation is removed in variant II.

5. Construction of the machine learning algorithm: Following the idea proposed by (Moustapha *et al.*, 2008) for the detection of sensor faults [26], our detector learns the fault-free instruction flow of RSA. After learning, it can detect faults that break this flow. Hence, we need a machine learning algorithm that can incorporate previously executed instructions. We use an RNN for this task. The construction of an RNN consists of three tasks: network design, training and evaluation. In the network design task, we select network parameters such as types of layers, numbers of layers, number of recurrent cells (also referred as RNN cells), etc. During the training task, we train the RNN using a training set of correct decryptions. Lastly, in evaluation, we determine the performance of the trained RNN by using a test set, which contains faulty decryptions.

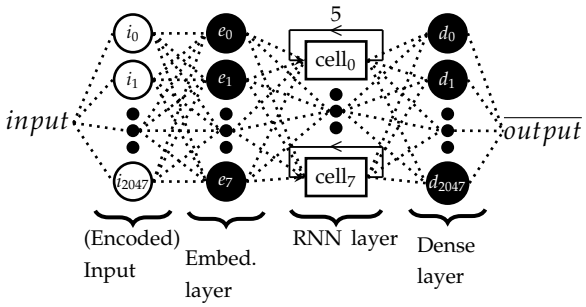

Fig. 1. RNN used in this work

6. Implementation of the detector: Our detector monitors the executed instructions. Therefore, it is implemented as a hardware module next to the CPU. To minimize the area overhead, we construct our RNN as hardware optimal as possible.

III. RNN-Based Detector

This section details Steps 5 and 6 from Section II.

A. Construction of Machine Learning Algorithm

1. RNN design: We design our RNN to compute the expectance probability of the current fetched instruction by using the five previously fetched ones. Figure 1 shows the layers and the dimensions of the neural network. The neural network contains three layers (i.e., embedding, RNN and dense layer).

The input to the first layer are the fields of the instruction that determine the instruction type. For the RISC-V, they are opcode, f3 and one bit of f7. Together they have a length of 11 bits. We encode this by using one-hot encoding. The first layer of the network is the embedding layer. This layer reduces the size of the one-hot vector to eight elements. The outputs of the embedding layer are connected to the RNN layer. This RNN layer processes the last five instructions and outputs a vector to the last layer. The last layer (i.e., the dense layer) is used to make the decisions. The neurons in this layer produce the expectance value of their corresponding instructions (e.g., neuron 659 gives the expectance value of instruction 659). We only look at the output of the currently fetched instruction as it determines its likelihood of occurrence; an unlikely instruction indicates a fault.

2. RNN training: The training phase of the detector consists of five steps. The aim of the *first step* is to create binaries of a software RSA decryption written in C. We use one implementation based on CRT and one without. Any step described hereafter applies to both cases. Both implementations use random keys, plain-, and ciphertexts. Moreover, they use square-and-multiply (SAM) algorithm for faster exponentiations. The implementation with CRT uses also the extended Euclidean algorithm for calculating modular inverses. To create the binaries, we use the riscv_gcc (version 7.1) compiler [27].

In the *second step*, we use the generated binaries to initialize the instruction memory of the RISC-V processor. This processor is implemented in RTL and is

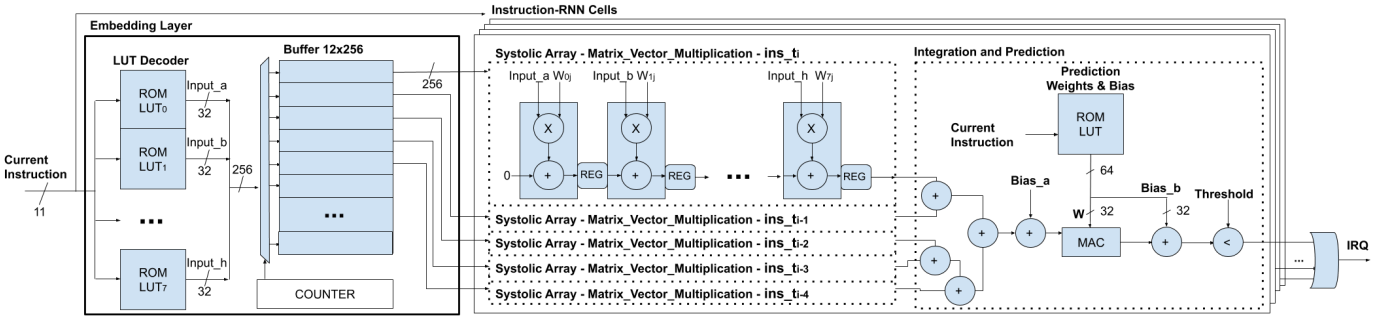

Fig. 2. Hardware architecture of the RNN-based detector

part of a system-on-chip (processor, cache memories and peripherals). In the *third step*, we compile and simulate this RTL code with the testbench using the QuestaSIM simulator [28]. During the simulation, processor fetches instructions from the instruction memory and executes them. The simulator saves this sequence of executed instructions into a text file.

In the *fourth step*, we construct two datasets using the text file with executed (non-faulty) instructions. These sets are the *training set* and *validation set*. Finally in the fifth step, we use the *training set* to train the RNN with the aim of predicting the next instruction when previous five are supplied. We then use the *validation set* to calculate a threshold value for the expectance. This $conf_{thr}$ is the lowest expectance value of all the instructions in the *validation set*. Later during runtime, if an instruction has a lower expectance value, the detector considers it as faulty.

3. RNN evaluation: The first two steps of the evaluation phase are identical to the training phase. However, in the *third step*, we modify the testbench to inject faults to the instructions during simulation. To do this, the simulator randomly glitches the instruction memory using one of our fault models (see Section II). The simulator saves the executed instruction sequence into a text file. Finally in the *fourth step*, we construct a dataset named *test set* that contains these faulty decryptions. Using this *test set*, with the trained RNN and calculated $conf_{thr}$, we evaluate the detection rate of faults.

B. Implementation of the Detector

The hardware implementation of detector is shown in Figure 2. It consists of two major components: the embedding layer (containing the input encoding and embedding layer) and the Instruction-RNN cell (containing the RNN cells and dense layer).

The function of the embedding layer is to generate eight input numbers to the RNN cells when an 11 bit instruction ($id \in \{0, 1, \ldots, 2047\}$) is provided (see also Figure 1). We implement this layer as a lookup table (LUT). The hardware implementation also contains a decoded instruction buffer, which is used to store data to keep up with the instruction fetch speed of the processor; this is explained in more details later.

The second component is the Instruction-RNN cell. It is further divided into two sub-components: five systolic arrays implementing the RNN cells and Integration and Prediction unit that combines the outputs of the RNN cells, which makes a prediction of the correctness of the currently fetched instruction. The systolic array architecture [29] implements the vector matrix multiplications of the RNN cells. In total, five systolic array units are used where each one contains eight multiplication-and-accumulation (MAC) elements. To optimize the RNN cells, the nonlinear operations are removed at the cost of a loss in accuracy. In addition, we unroll the cells and pipeline them. To do this, we rearrange the RNN function (that gives the probabilities for instructions at time t_{i+1} using five previous instructions) as follows:

$$\begin{aligned}\overline{h}_{t_i} = \boldsymbol{W'}\overline{ins}_{t_i} + (\boldsymbol{ZW})'\overline{ins}_{t_{i-1}} + (\boldsymbol{Z}^2\boldsymbol{W})'\overline{ins}_{t_{i-2}} \\ + (\boldsymbol{Z}^3\boldsymbol{W})'\overline{ins}_{t_{i-3}} + (\boldsymbol{Z}^4\boldsymbol{W})'\overline{ins}_{t_{i-4}} + \overline{B}, \qquad (1)\end{aligned}$$

where $\boldsymbol{W}$ is the weight matrix for the feedforward input, $\overline{ins}_{t_i}$ (indicated by *Input*s in Figure 2) is the embedded layer output vector for the instruction at time t_i, $\boldsymbol{Z}$ is the weight matrix for the feedback input, and $\overline{B}$ is the collective bias vector (indicated by Bias_a in Figure 2). This equation enables us to precompute and store all matrices and vectors except $\overline{ins_t}$'s. This reduces the number of multiplications and additions almost four times.

The Integration and Prediction unit sums up the results of the five systolic arrays and additionally implements the dense layer used for final prediction. The dense layer contains 2048 neurons, where each neuron corresponds to the expectance probability of an instruction. In our hardware implementation, we only use a single neuron. When instruction at time t_{i+1} becomes available, we load its corresponding neuron weights from a LUT (indicated by Prediction Weights & Bias on the figure). Secondly, we remove the sigmoid activation function as it affects only the output range. As we only compare the output to a threshold confidence value, no accuracy is lost here. The MAC (for vector multiplication) and the subsequent adder in this unit implement the last neuron. When an attack is detected, a signal is sent to the CPU as an interruption request (IRQ).

The last important point is that our Instruction-RNN-cell outputs a result each 8 clock cycles. As each cycle a new instruction is fetched, we place 7 Instruction-RNN-cells in parallel to be able to process all instructions. To prevent loss of data, we add the aforementioned buffer

TABLE I
ACCURACY EVALUATION OF THE RNN-BASED DETECTOR

		fault		*decryption*		*security*	
fault model	**#faults (f)**	**CRT**	**non-CRT**	**CRT**	**non-CRT**	**CRT**	**non-CRT**
1	$f = 1$	0.37	0.27	0.71	0.54	0.75	0.88
	$f > 1$	0.67	0.60	0.73	0.68	0.83	0.94
2	$f = 1$	0.62	0.52	0.79	0.69	0.82	0.94
	$f > 1$	0.90	0.82	0.92	0.85	0.94	0.98
3	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
4-I	$f = 1$	0.91	0.97	0.93	0.97	0.94	1.00
	$f > 1$	0.99	1.00	0.99	1.00	1.00	1.00
4-II	$f = 1$	0.92	0.97	0.95	0.97	0.96	1.00
	$f > 1$	0.99	1.00	0.99	1.00	0.99	1.00

to the embedding layer, which stores 12 instruction features. Moreover, we replace all floating point numbers in the design with 32-bit fixed numbers.

IV. EXPERIMENTAL RESULTS

This section describes the experiment and results.

A. Experimental Setup

We implemented the C-based RSA implementations using 12-bit keys (without loss of generality) to speedup simulations. The RNN is trained using the adam optimizer [30], categorical crossentropy, and accuracy as a metric. A dropout of 0.1 (normal and recurrent) was used to avoid overfitting. Our *training set* consists of 750 correct decryption operations, whereas the *validation set* consists of 250. During training, a batch and epoch size of 100 are used. In the end, we obtained $conf_{thr}$ values of 4.3900 for the CRT and 10.9570 for the non-CRT case. In the evaluation phase, we modified the network based on the optimization discussed in the previous section such as using 32-bit fixed point decimals and removing the sigmoid from the dense layer neurons. Subsequently, we used 2000 (faulty) decryptions per fault injection model for evaluating the fault detection performance of our detector. Lastly, we also used 10,000 correct decryptions that were not parts of the *training set* and *validation set* to test the functionality of the detector. We implemented both training and evaluation using Python with Tensorflow [31] and Keras [32].

We evaluated the overhead by comparing the area of our detector against Ariane core [33]. Ariane is a RISC-V processor with Linux support that is suitable for high-end applications (e.g., smartphones). The complete hardware design was synthesized for the device 10AS066N3F40ELG from the ARRIA 10 family [34] of the Intel FPGA platform.

B. Results

First, we investigated the detector efficiency against correct decryptions. The detector successfully classified 10,000 correct operations all as non-faulty for both CRT and non-CRT implementations, which means we achieved 0% false positive rate. Next, we investigated the efficiency of the detector by creating faults in the binary using the four fault injection models presented in Section II and evaluated how often Bellcore and Bao based attacks were successful. Table I presents the results. In the table, we arranged the results into three classes: fault, decryption, and security. The fault column contains the rate of faulty decryptions that were detected. The decryption column includes the ratio of decryptions we can protect (i.e., fault detection rate plus the cases where the fault did not change the result of the decryption and thus, the attacker cannot exploit). The security column contains the ratio of traces that could not be attacked (i.e., decryption rate plus the faulty decryptions that neither Bellcore nor Bao threats can exploit. Note that the Bellcore is not applied to non-CRT RSA). For each fault model, two cases were considered: a single faulty instruction $f = 1$ and multiple faulty instructions $f > 1$.

The results, which are similar for both RSA implementations, show that faults that change instructions (which are the most effective faults as used in fault models 3, 4-I, and 4-II) can be detected very well and that the detector provides almost 100% security. For fault models 1 and 2, the detection of faults is much lower as the injected faults often glitched the data parts which are not analyzed by the neural network. In contrast to faults affecting data fields, our analysis showed a high accuracy for bit and byte-level faults in the control fields of an instruction. Overall, the average decryption/security rate per fault model is around 60-80% for these fault models. However, these numbers are far from a perfect detection, and therefore, this is the main limitation of our detector.

The synthesis results show that our detector equals to 15% (34,785 LC combinationals) of the combinational logic and 1.5% (8211 LC registers) of the registers of the Ariane core, which corresponds to 5.7% of the total area of Ariane. In addition, our solution requires 80kB of memory to realize the LUTs related to network weights.

V. CONCLUSION

In this study, we proposed a fault detector for software RSA implementations. To test its effectiveness, we proposed a collection of fault injection models. Our results show that our model works especially well against faults that change an instruction to another. Additionally, with only changing the network weights, we were able to protect two different implementations, which shows the flexibility of our detector. Finally, the hardware implementation is shown to be practical as it is less than 6% of a high-end processor. Future work aims to improve the accuracy of the detector especially in detecting bit and byte faults, by evaluating the instructions together with other micro-architecture information; such as memory addressing, register controls, or branch decisions.

VI. ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 722325.

References

[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[2] F. Wang and V. D. Agrawal, "Single event upset: An embedded tutorial," in *21st International Conference on VLSI Design (VLSID 2008)*. IEEE, 2008, pp. 429–434.

[3] T. W. Houston, "Single event upset hardened memory cell," May 18 1999, uS Patent 5,905,290.

[4] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2002, pp. 2–12.

[5] J.-M. Schmidt, M. Hutter, and T. Plos, "Optical fault attacks on aes: A threat in violet," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2009, pp. 13–22.

[6] N. Selmane, S. Guilley, and J.-L. Danger, "Practical setup time violation attacks on aes," in *2008 Seventh European Dependable Computing Conference*. IEEE, 2008, pp. 91–96.

[7] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *IEEE Symposium on Security and Privacy*, vol. 5, 2003.

[8] M. Agoyan, J.-M. Dutertre, A.-P. Mirbaha, D. Naccache, A.-L. Ribotta, and A. Tria, "How to flip a bit?" in *2010 IEEE 16th International On-Line Testing Symposium*. IEEE, 2010, pp. 235–239.

[9] X. T. Ngo, J.-L. Danger, S. Guilley, T. Graba, Y. Mathieu, Z. Najm, and S. Bhasin, "Cryptographically secure shield for security ips protection," *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 354–360, 2016.

[10] L. Anghel and M. Nicolaidis, "Cost reduction and evaluation of a temporary faults-detecting technique," in *Design, Automation, and Test in Europe*. Springer, 2008, pp. 423–438.

[11] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented aes: effectiveness and cost," in *Proceedings of the 5th Workshop on Embedded Systems Security*. ACM, 2010, p. 7.

[12] R. Karri, K. Wu, P. Mishra, and Y. Kim, "Fault-based side-channel cryptanalysis tolerant rijndael symmetric block cipher architecture," in *Proceedings 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE, 2001, pp. 427–435.

[13] R. Karri, G. Kuznetsov, and M. Goessel, "Parity-based concurrent error detection of substitution-permutation network block ciphers," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2003, pp. 113–124.

[14] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on rsa with crt: Concrete results and practical countermeasures," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2002, pp. 260–275.

[15] A. Boscher, H. Handschuh, and E. Trichina, "Fault resistant rsa signatures: Chinese remaindering in both directions." *IACR Cryptology ePrint Archive*, vol. 2010, p. 38, 2010.

[16] A. Shamir, "Method and apparatus for protecting public key schemes from timing and fault attacks," Nov. 23 1999, uS Patent 5,991,415.

[17] D. Vigilant, "Rsa with crt: A new cost-effective solution to thwart fault attacks," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2008, pp. 130–145.

[18] C. Giraud, "An rsa implementation resistant to fault attacks and to simple power analysis," *IEEE Transactions on computers*, vol. 55, no. 9, pp. 1116–1120, 2006.

[19] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[20] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[21] A. Waterman and K. Asanovic, "The risc-v instruction set manual-volume i: User-level isa-document version 2.2," *RISC-V Foundation (May 2017)*, 2017.

[22] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1997, pp. 37–51.

[23] F. Bao, R. H. Deng, Y. Han, A. Jeng, A. D. Narasimhalu, and T. Ngair, "Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults," in *International Workshop on Security Protocols*. Springer, 1997, pp. 115–124.

[24] A. Barenghi, C. Hocquet, D. Bol, F.-X. Standaert, F. Regazzoni, and I. Koren, "Exploring the feasibility of low cost fault injection attacks on sub-threshold devices through an example of a 65nm aes implementation," in *International Workshop on Radio Frequency Identification: Security and Privacy Issues*. Springer, 2011, pp. 48–60.

[25] A. Barenghi, G. Bertoni, E. Parrinello, and G. Pelosi, "Low voltage fault attacks on the rsa cryptosystem," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2009, pp. 23–31.

[26] A. I. Moustapha and R. R. Selmic, "Wireless sensor network modeling using modified recurrent neural networks: Application to fault detection," *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 5, pp. 981–988, 2008.

[27] "The risc-v embedded gcc," Jul 2017. [Online]. Available: https://gnu-mcu-eclipse.github.io/toolchain/riscv/

[28] "Questa® advanced simulator." [Online]. Available: https://www.mentor.com/products/fv/questa/

[29] L. D. Medus, T. Iakymchuk, J. V. Frances-Villora, M. Bataller-Mompeán, and A. Rosado-Muñoz, "A novel systolic parallel hardware architecture for the fpga acceleration of feedforward neural networks," *IEEE Access*, vol. 7, 2019.

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[31] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

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

[33] J. Balkind, "Openpiton + ariane : The first open-source , smp linux-booting risc-v system scaling from one to many cores," 2019.

[34] "Intel Arria 10 FPGAs," accessed at 23-10-2019. Available at: https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html.