Using Hopfield Networks to Correct Instruction Faults

Köylü, T.C.; Fieback, M.; Hamdioui, S.; Taouil, M.

DOI
10.1109/ATS56056.2022.00030

Publication date
2022

Document Version
Final published version

Published in
2022 IEEE 31st Asian Test Symposium (ATS)

Using Hopfield Networks to Correct Instruction Faults

Troya Çağıl Köylü, Moritz Fieback, Said Hamdioui and Mottaqiallah Taouil
Quantum & Computer Engineering Department
Delft University of Technology
Delft, the Netherlands
{T.C.Koylu, M.C.R.Fieback, S.Hamdioui, M.Taouil}@tudelft.nl

***Abstract*—Fault injection attacks pose an important threat to security-sensitive applications, such as secure communication and storage. By injecting faults into instructions, an attacker can cause information leakage or denial-of-service. Hence, it is important to secure the sensitive parts not only by detecting faults in the executed instructions but also by correcting them. In this work, we propose a hardware detection and correction module based on Hopfield networks. Our module is connected to the instruction buffer and validates all fetched instructions. In case faults are detected, faulty instructions are replaced by corrected ones. Experimental results on a small RISC-V processor and two RSA implementations show that we achieve near perfect detection and around 70% accurate correction with 9% area overhead. This correction rate is enough to secure some implementations for all considered attacks.**

***Index Terms*—statistical error correction, Hopfield networks, fault injection, hardware security, machine learning**

I. Introduction

Fault injection attacks are important threats to microprocessors, as they can cause disruptions in operation and even leak secret data [1]. It is relatively easy for an attacker to inject these faults, as simple techniques such as clock glitching or voltage underfeeding can be used [2], [3]. This has been shown for widely-used cryptosystems in numerous instances such as RSA [4], where fault injection leads to faulty instructions [5], [6]. Furthermore, recently it was demonstrated that it is possible to underfeed the voltage remotely to a line of secure Intel chips, resulting in leaking secrets from a Chinese remainder theorem (CRT)-based RSA implementation [7]. Although it is important to detect these faulty instructions and prevent faulty results from reaching the attackers, doing so should not result in a denial-of-service attack. Hence, mechanisms to not only *detect* but also *correct* faulty instructions are crucial to sustain security-sensitive applications, especially crypto implementations that are widely employed.

Solutions that address the detection and correction of instruction faults typically introduce some sort of redundancy. The redundancy can be added in time or hardware. Redundancy in time is typically implemented on the software level, e.g., by repeating instructions that are prone to faults [8]. This increases the execution time of the program and does not guarantee that all faults will be corrected. Hardware approaches typically introduce error correcting codes [9], [10], signature comparisons [11], or duplicate/triplicate hardware that is prone to faults [12], [13]. Error correcting codes can only detect and correct a limited number of bit-flips and as such do not protect all instructions in security-sensitive applications, just like signature comparisons. Duplicating or even triplicating the hardware is very expensive in terms of area and power consumption, which prohibits the use in resource-constrained devices, such as IoT. Therefore, for security-demanding applications, there is a need for a scheme with high error detection/correction capability and low area overhead.

To address this need, we present a statistical error correction scheme that is based on Hopfield networks. Our scheme is able to (i) detect and (ii) correct erroneous instructions before they are executed, thereby preventing (i) information leakage and (ii) denial-of-service. In summary, our contributions in this work are as follows:

- Proposal of an efficient and effective statistical error correction scheme based on Hopfield networks to detect and correct instruction faults.
- Efficient hardware design of the proposed error correction scheme for general-purpose processors.
- Demonstration of the effectiveness of the proposed scheme by fault attack experimentation on two RSA implementations for different fault attack models.

The rest of this paper is organized as follows. Section II describes the motivation and our proposed error correction scheme. Section III presents an efficient hardware design of the scheme, while Section IV validates it experimentally. Finally, Section V concludes the paper.

II. Methodology

In this section, we first describe the concept behind our detection and correction scheme. Thereafter, we describe the application and threat model. Finally, we show how Hopfield networks can be used for the scheme.

A. Concept

Our fault detection and correction scheme is a hardware module that resides between the instruction buffer

2377-5386/22/$31.00 ©2022 IEEE
DOI 10.1109/ATS56056.2022.00030

2022 IEEE 31st Asian Test Symposium (ATS) | 978-1-6654-7227-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/ATS56056.2022.00030

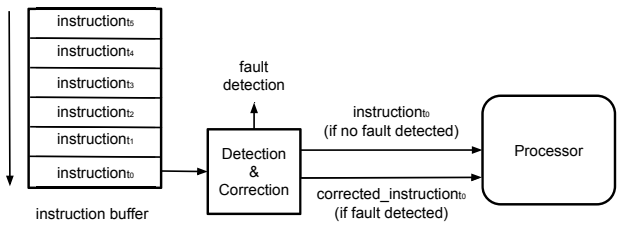

Fig. 1: Functional Overview of the Proposed Scheme

and the processor. Its functionality is shown in Figure 1.

As indicated in the figure, our module verifies the instructions that the processor fetches. The module is application dependent and a fault in an instruction causes an unexpected instruction for the application. During instruction verification, the instruction is forwarded to the processor without modifications in case no fault has been detected. On the other hand, when the detection and correction module detects a fault in an instruction, it raises the fault flag and sends the corrected instruction to the processor instead. Note that this solution is not disruptive to the implementation of the processor and hence, does not require costly modifications to it.

B. Application and Threat Model

In this work, we focus on commonly employed software implementations of crypto cores [14] as security-sensitive application. The reason is twofold: (i) given the inputs (e.g., key and ciphertext), cryptosystems typically generate a deterministic output. This makes it possible to check for application reliability and denial-of-service in general under the influence of a fault attack, by comparing the golden output with the obtained output; (ii) there are mathematical models that describe how secret key information can be leaked from a faulty output and hence, it allows us to evaluate the proposed detection and correction scheme.

The RSA cryptosystem [15] presents a very suitable case study. Namely, there are multiple ways to implement the algorithm which results in different vulnerabilities. A commonly used algorithm to make the RSA modular calculation faster is using CRT [16]. This implementation however is vulnerable to the Bellcore threat [17], which may leak the entire secret key when correct and faulty decryption results are compared. Another popular vulnerability is based on Bao's threat [18], which is also applicable to RSA implementations without CRT. With each successfully inserted fault, one bit of the secret key may be leaked.

The Bellcore and Bao threats require fault injections to specific values during the calculation. Many fault attacks on instructions can be used to exploit these vulnerabilities. To evaluate the strength of the detection and correction scheme, we use the following fault attack models [19]: *bit-level*, *byte-level*, *branch-to-opposite*, and *instruction-to-instruction I/II* fault models.

The bit-level fault model comprises a single bit flip in an instruction. This can be caused naturally by radiation [1] or by sophisticated attack means such as laser-based fault injection [20] and Rowhammering [21]. The byte-level fault model comprises corrupting one byte of the instruction. This can be caused by less sophisticated attack means, such as exposing the chip to ultraviolet light [22]. The branch-to-opposite fault model comprises changes in branch instructions to their opposite ones (e.g., branch equal to branch not equal), effectively reverting the branch behavior. This was for example achieved in [5], where the authors used voltage underfeeding to successfully break RSA. Finally, instruction-to-instruction I/II fault models are extensions of the previous one, where an instruction can be changed into any other valid instruction. Variant I prevents another instruction to become a branch to avoid many crashes, while variant II does not come with this limitation. These two fault models can only be realized by very sophisticated means of attack, such as using multiple lasers at once.

C. Hopfield Network-Based Instruction Fault Detection and Correction

The implementation of our detection and correction scheme is based on Hopfield networks. The original Hopfield network is a basic memory structure that recalls patterns. Its simplicity and recalling property is the main reason why we use Hopfield networks, instead of complex networks like a convolutional neural network. A simple example with one iteration is illustrated in Figure 2. The memory in the example contains six neurons (n_0 to n_5) and hence, it can recall patterns of length six. For simplicity, we assume that the patterns consist of bipolar bits, i.e., $x \in \{-1, 1\}^6$. When a new pattern x_{new} is provided for evaluation, the network tries to reconstruct it with the resembling patterns that were previously learned. Initially, the state equals the input, i.e., $\xi_0 = x_{new}$. Thereafter, the new state is obtained by multiplying the current state ξ_0 with the weight matrix $\boldsymbol{W}$, resulting in ξ_1. In general, the state update formula for t iterations equals the following [23]:

$$\xi_{t+1} = \text{sgn}(\boldsymbol{W}\xi_t). \tag{1}$$

Here, sgn represents the sign function with the output either equal to -1 (if the argument is negative) or 1 (if it is positive). The iterations end when the new state equals the current one, i.e., $\xi_{t+1} = \xi_t$. Furthermore, the weight matrix is simply obtained from the dot product of the learning patterns, i.e., $\boldsymbol{W} = \sum_{i=0}^{N-1}(x_i x_i^T)$, where $x_i | i \in [0, N)$ are the learned patterns.

The main issue with the example Hopfield network in Figure 2 is its very limited memory capacity. According

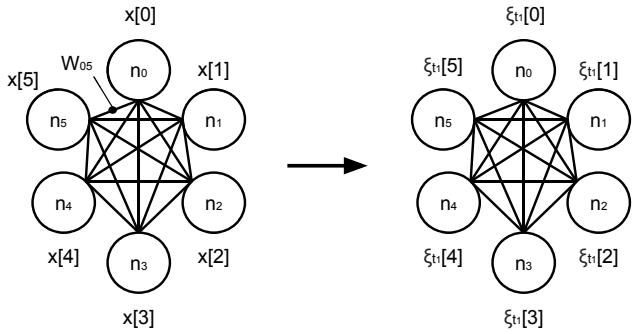

Fig. 2: Sample Hopfield Network over a State Iteration

to the formula provided in [24], this Hopfield network is expected to only memorize 0.84 patterns. The capability of storing patterns has to be increased, which can be achieved by including non-linear operations inside the neurons. This enables both a capacity increase and an improved ability to distinguish between close patterns. This is indicated by the following formula [25], which is a modification of Equation 1:

$$\xi_{t+1}[l] = \text{sgn}[\sum_{i=0}^{N-1} F(x_i^T \xi_t^{(l^+)}) - \sum_{i=0}^{N-1} F(x_i^T \xi_t^{(l^-)})]. \quad (2)$$

Here, $\xi_t^{(l^+)}$ and $\xi_t^{(l^-)}$ only differ in bit l, where $\xi_t^{(l^+)}[l] = 1$ and $\xi_t^{(l^-)}[l] = -1$. F is the aforementioned nonlinear function that increases the capacity. If $F(a) = a^2$, the simple Hopfield network is obtained. When the exponent is higher, the recall capability of the neurons and thus the overall memory capacity increases in a nonlinear fashion [26]. For instance, when $F(a)$ is changed from a^2 to a^3, the expected number of stored patterns with the same six neurons increases from 0.84 to 3.35.

With nonlinear Equation 2, it is theoretically possible to store all unique instructions of a program by using 32 or 64 neurons - equal to the typical instruction size. The unique instructions of the RSA implementation can be extracted from the binary. The Hopfield network learns (or stores) these instructions. At runtime when the RSA implementation is executed, fetched instructions are validated and corrected using the learned instructions (see Equation 2).

There are however two challenges in realizing this. The first and the main challenge is the hardware cost of implementing the nonlinear function $F(a)$. The second is the iterative nature of Equation 2 to reach convergence, which makes the hardware implementation more difficult due to the potential need for multiple cycles. Note that the convergence state can also be an invalid instruction. This is typically not a problem when stored patterns are images and the reconstructed image only differs in a couple of pixels. However in our case, even a single bit difference in the corrected instruction can result in crashes (when a faulty instruction is corrected to an invalid one) or significantly different results (e.g., when loading a value from an incorrect address or register).

To solve both issues, we analyze Equation 2 in more depth. Let's assume a very large value for the exponent K in $F(a) = a^K$ to increase the performance of the network. This results in the following equation:

$$\xi_{t+1}[l] = \text{sgn}[\sum_{i=0}^{N-1} (x_i^T \xi_t^{(l^+)})^K - \sum_{i=0}^{N-1} (x_i^T \xi_t^{(l^-)})^K]. \quad (3)$$

Let us consider two scenarios for this equation. The first case is when $\xi_t = x_{\hat{i}}$ (i.e., the current instruction equals the stored instruction $\hat{i}$). In this case, $\forall l$ the following holds: $x_i^T \xi_t^{(l^+)}$ will dominate the summation if $\xi_t = \xi_t^{(l^+)}$ and $x_i^T \xi_t^{(l^-)}$ otherwise; and determine the bit as $l = x_{\hat{i}}[l]$. In the end, $\xi_{t+1} = x_{\hat{i}}$ will hold.

The second case is when ξ_t differs from $x_{\hat{i}}$ by one bit, due to for example a fault at $\hat{l}$. Then, $l = x_{\hat{i}}[l]$ holds $\forall l \neq \hat{l}$. On the other hand, the sign will be reversed for $\xi_{t+1}[\hat{l}]$, as $x_i^T \xi_t^{(l^+)}$ will dominate the summation if $\xi_t \neq \xi_t^{(\hat{l}^+)}$ and $x_i^T \xi_t^{(\hat{l}^-)}$ otherwise. This will make ξ_{t+1} and $x_{\hat{i}}$ differ in only one bit. Thus, correcting as $\xi_{t+1} = x_{\hat{i}}$ will be valid.

From these, we conclude from the ideal case that when K is very large, each bit of ξ_{t+1} is heavily influenced by the most similar stored instruction $x_{\hat{i}}$. Hence, it is meaningful to make the instruction correction as $\xi_{t+1} \approx x_{\hat{i}}$. This eliminates the need for multiple iterations to reach convergence (by equating the new state simply to one of the stored instructions). In addition, it enables us to simplify the calculations into a bitwise comparison with all stored instructions and select the most similar one.

III. Hardware Implementation

The implementation of our fault detection and correction scheme is based on the observations made in Section II-C. We integrate it with a 32-bit RISC-V processor [27].

The architecture of the detection and correction scheme is provided in Figure 3. As shown in the figure, the architecture consists of three stages: Stage 1 - comparison, Stage 2 - calculation, and Stage 3 - verification. In the *comparison stage*, the current instruction (denoted as ins_{t_0}) is compared with all stored unique instructions (denoted as $st_i | i \in [0, N)$). The comparison is done with modified XOR gates, which output the total number of bits that are different. As the maximum difference can at most be 32, five bits are needed.

The *calculation stage* aims to find the instruction ID (i.e., the storage address of the instruction in our module) with the minimum difference. To this end, this stage uses a tree-like structure of *min unit*s. These units take

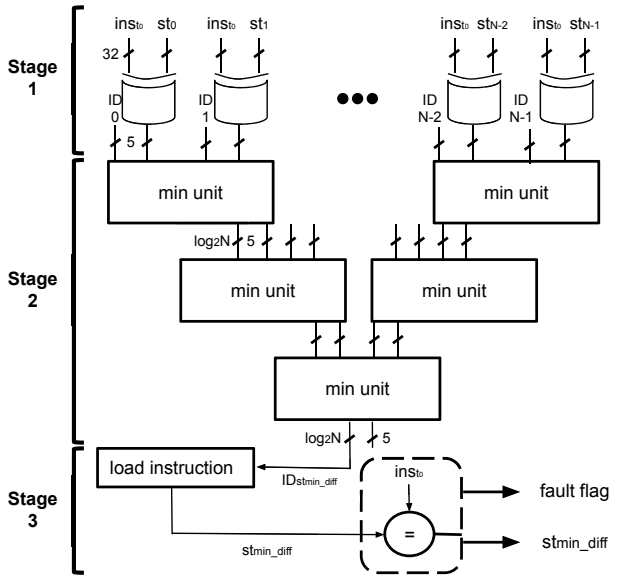

Fig. 3: Hardware Architecture of the Detection & Correction Module

four inputs: two difference values (represented with 5 bits) and two corresponding IDs (represented by $\log_2 N$ bits). This unit simply forwards the value and the ID of the smaller difference. Naturally, the depth of the tree depends on the number of stored instructions N. The end product of this stage is the ID of the most similar stored instruction $st_{\text{min\_diff}}$.

The third and the final *verification stage* determines if there is a fault. It accomplishes this by first loading $st_{\text{min\_diff}}$ using the ID output of Stage 2. Then, it compares to see if $ins_{t_0} = st_{\text{min\_diff}}$. If they are equal, it sets the fault flag to 0 and to 1 otherwise. In both cases, it forwards $st_{\text{min\_diff}}$ to the processor, which is the same instruction in the no fault detected case and the corrected version of the instruction in the fault detected case.

IV. Validation

In this section, we describe the experimental setup and validate the effectiveness of our proposed scheme.

A. Experimental Setup

The experimental setup is based on the one presented in [19]. We used random keys for the C-based RSA decryption implementations with and without CRT. We first extracted the unique instructions from their corresponding binaries. This resulted in 120 unique instructions for the CRT and 48 for the non-CRT implementation. Note that this extraction process does not consider speculative or out-of-order execution. However, they do not affect our module, as instructions are checked individually without considering their order.

In this work, we conduct two sets of experiments. In the first one, we investigate the performance of our simplified Hopfield network based on bitwise comparisons against other Hopfield realizations with higher exponentiation (see Section II-C). This experiment is conducted in Python using the unique instructions of the non-CRT implementation. In the second experiment, we investigate the error detection and correction performance of our hardware scheme. Per fault model (see Section II-B), we conduct 1000 RSA decryptions. This is performed for both RSA implementations.

We carried out all hardware simulations for the second set of experiments using the QuestaSIM simulator [28], where our Hopfield-based detection and correction scheme is integrated into a RISC-V processor. We inject the faults into the instruction buffer during each RSA decryption run. At each run, we inject faults in one up to four different instructions based on the selected fault model (see Section II-B). These faults are injected randomly at run-time. Note that the faulty instructions that are sent to the processor for execution are either accurately or inaccurately corrected by the detection and correction scheme.

Finally, we evaluated the hardware overhead of our module by synthesizing it for the FPGA device XC7K325TFFG900-2 from the Kintex-7 family [29] and comparing it with the RI5CY core [30]; RI5CY is a small scale RISC-V processor and is synthesized on the same FPGA device.

B. Accuracy of Simplified Hopfield Network

In this experiment, we compare the performance of our simplified Hopfield network based on bitwise comparisons with the non-simplified $F(a) = a^2$ standard Hopfield network, $F(a) = \exp(a)$ [24], and $F(a) = a^8$. We conduct three trials in this experiment.

In the first trial, we test the performance of the different Hopfield networks without injecting any faults. Here we want to check whether they can store the instructions properly and analyze how many iterations are needed for convergence. For $F(a) = a^2$, $F(a) = \exp(a)$, and $F(a) = a^8$, we limit the number of iterations to 10 in order to avoid infinite loops of non-convergence. In the second trial, we exhaustively inject bit-flips in every bit of the unique instructions, one at each time (i.e., there are in total $48 \times 32 = 1536$ runs). In the third trial, we corrupt the bytes of the unique instructions to create 20 faulty instances (i.e., there are $48 \times 20 = 960$ runs).

Table I presents the results of the three trials; it shows how many instructions are corrected in an accurate and inaccurate manner, as well as the average number of iterations required to reach convergence.

As can be observed from the table, our simplified Hopfield network significantly outperforms the other realizations in terms of accurate corrections and required number of iterations. In addition, its hardware implementation is much cheaper.

TABLE I: Accuracy of Hopfield Networks

trial	method	accurate correction	inaccurate correction	average iterations
1	bitwise	48	0	1.0
	a^2	1	47	4.54
	$\exp(a)$	35	13	1.31
	a^8	34	14	1.33
2	bitwise	1488	48	1.0
	a^2	32	1504	4.63
	$\exp(a)$	1089	447	2.26
	a^8	1055	481	2.26
3	bitwise	782	178	1.0
	a^2	20	940	4.66
	$\exp(a)$	579	381	3.02
	a^8	537	423	3.18

TABLE II: Detection Performance

		detection		*decryption*		*security*	
fault model	**#faults (f)**	**CRT**	**non-CRT**	**CRT**	**non-CRT**	**CRT**	**non-CRT**
1	$f=1$	0.97	1.00	0.98	1.00	0.98	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00
2	$f=1$	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
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	$f=1$	0.94	0.93	0.98	0.95	0.99	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00
4-II	$f=1$	0.98	0.99	1.00	1.00	1.00	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00

C. Validation of Error Detection and Correction Scheme

First, we tested our scheme with non-faulty decryptions. For this, we used 10,000 non-faulty decryptions. Our scheme did not raise any false alarms in these runs. Next, we tested our module against faulty runs. Our module successfully detects a fault if it raises at least one fault flag during that run. Table II presents the detection results.

In the table, the results are presented in three classes, just as in [19]: detection, decryption (coverage), and security (coverage). Detection is the percentage of faulty runs that our module detected. The decryption column also includes the undetected cases where the decryption output is still correct, making it impossible for an attacker to exploit such faults. The security column further includes the faulty output cases that Bellcore and Bao threats were not able to exploit. Note that Bellcore threat is only applicable to the CRT implementation, while Bao is applicable to both. Moreover, the table further differentiates cases where only one instruction is faulty (i.e., $f = 1$) and cases where more than one instruction is faulty (i.e., $f > 1$). As the results show, we achieve perfect or near perfect detection in all cases except for fault model 4-I where $f = 1$. Note that this fault model has constraints that make the detection harder; instructions are changed into other valid instructions, while branches are protected. Overall, the decryption and security coverage is even higher than detection, making it very hard for an attacker to leak information while our module is active. The non-CRT case is even fully secure against Bao's attack.

Next, we test the correction performance of our scheme. The results are presented in Table III. This table

TABLE III: Correction Performance

		correction		*operational*	
fault model	**#faults (f)**	**CRT**	**non-CRT**	**CRT**	**non-CRT**
1	$f=1$	0.94	0.96	0.94	0.97
	$f>1$	0.81	0.92	0.91	0.97
2	$f=1$	0.67	0.77	0.78	0.84
	$f>1$	0.31	0.63	0.59	0.75
3	$f=1$	1.00	1.00	1.00	1.00
	$f>1$	1.00	1.00	1.00	1.00
4-I	$f=1$	0.45	0.46	0.49	0.48
	$f>1$	0.35	0.30	0.60	0.41
4-II	$f=1$	0.66	0.76	0.69	0.79
	$f>1$	0.49	0.58	0.67	0.65

presents the results in two cases: (i) correction and (ii) operational (coverage). The correction column presents the ratio of faulty runs where the scheme corrected the instructions in an accurate manner. The operational column includes all cases where the output is correct, i.e., cases where the correction was successful but also some cases where it was not.

As can be observed from the table, the correction performance does not significantly vary per implementation, but does vary for different fault models. For the bit-level faults (fault model 1), our module attains >81% correction rate. When there is only one bit-flip, the success rate increases to >94%. Furthermore, our module attains a perfect correction rate for branch faults (fault model 3). The correction rate however drops for other fault models. In the majority of the cases for these fault models however, the correction rates are still acceptable, i.e., 70.3% on average. The operational coverage on the other hand reaches 77.7%. The performance particularly suffers when there are multiple faults during a single run or faults change an instruction to another valid instruction (especially in fault model 4-I). Both performance drops are to be expected: it is harder to correct when there are more faults in different or in the same instruction. It must be stressed that these cases are not as common as the others.

A final observation from Tables II and III is that our module has a significantly higher fault detection performance than the correction performance. This is because it is enough to raise one fault flag during a run to successfully detect a fault, while all faulty instructions should be accurately corrected to achieve a correct run. The latter is particularly challenging when instructions are changed with multiple bits, making them potentially closer to other stored instructions.

Finally, Table IV shows the area of the RI5CY core and the detection/correction schemes for the CRT and non-CRT implementations. Next to the resources required for the detector, the overhead relative to the RISC-V processor is also indicated in brackets. Both schemes have less than 9.2% overhead as compared to the RI5CY core in terms of LUT slices.

TABLE IV: Hardware Overhead

implementation	slice LUTs	slice registers	f7 MUX	f8 MUX	DSPs
RI5CY core	15857	8333	3373	1286	6
Scheme - CRT	1453 (9.2%)	0 (0.0%)	3 (0.1%)	0 (0.0%)	0 (0.0%)
Scheme - non-CRT	1338 (8.4%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)

V. Discussion and Conclusion

In this work, we presented an effective and efficient statistical error correction scheme based on Hopfield networks. We conclude this paper by discussing the following points.

- **Security and Generality:** Our scheme is shown to be able to detect and correct faulty instructions for two RSA implementations. Especially, it is able to reach a near 100% security coverage. Furthermore, our scheme can be used in general for any secure application. The only requirement is that the designer includes a sufficient number of XORs and a depth of min units (see Section III). As such, different applications can seamlessly be protected by our module, without requiring any hardware changes.
- **Comparison:** Our scheme improves upon the state of the art in different aspects (see Section I). Instruction repeating, error correction codes, and signature comparisons are not able to correct different types of faults. This is shown to increase the vulnerability for some cases [31]. In contrast, our scheme is shown to detect *and* correct various instruction faults with various amounts of bitflips (up to 11 for fault models 4-I/II). Next, only certain hardware TMR can outperform our scheme. Triplicating the instruction buffer is not sufficient, as a fault injected during the execution of a branch instruction can cause all three instruction buffers to receive erroneous proceeding instructions. Our scheme can still detect these faulty instructions, in case they are not a part of the stored unique instruction set. Thus, triplicating the whole core is the only option that guarantees the detection and correction of all faulty instructions, which certainly outperforms our scheme, albeit with a huge added cost.
- **Limitations:** The results show that our correction performance is not as high as our detection performance. The main issue causing this is that some faults cause instructions to be closer to other stored unique instructions. A way to alleviate this is to make the instructions as different as possible. Hence, a compiler that uses maximally different instructions to accomplish the same operation can increase the correction performance significantly.

References

[1] H. Bar-El *et al.*, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, pp. 370–382, 2006.

[2] F. Amiel *et al.*, "Fault analysis of dpa-resistant algorithms," in *FDTC*. Springer, 2006, pp. 223–236.

[3] N. Selmane *et al.*, "Practical setup time violation attacks on aes," in *EDCC*.

[4] R. Abid *et al.*, "An optimised homomorphic CRT-RSA algorithm for secure and efficient communication," *Personal and Ubiquitous Computing*, pp. 1–14, 2021.

[5] A. Barenghi *et al.*, "Low voltage fault attacks on the rsa cryptosystem," in *FDTC*. IEEE, 2009, pp. 23–31.

[6] J.-M. Schmidt *et al.*, *Optical and EM fault-attacks on CRT-based RSA: Concrete results*, 2007.

[7] K. Murdock *et al.*, "Plundervolt: Software-based fault injection attacks against intel sgx," in *S&P*. IEEE, 2020, pp. 1466–1482.

[8] A. Mokhtarpour *et al.*, "PB-IFMC: A selective soft error protection method based on instruction fault masking capability," in *CSICC*, 2020, pp. 1–9.

[9] S. Gupta *et al.*, "SHAKTI-F: A fault tolerant microprocessor architecture," in *ATS*, 2015, pp. 163–168.

[10] Y.-J. Ke *et al.*, "An integrated design environment of fault tolerant processors with flexible HW/SW solutions for versatile performance/cost/coverage tradeoffs," in *ITC-Asia*, 2017, pp. 162–167.

[11] N. Farazmand *et al.*, "FEDC: Control flow error detection and correction for embedded systems without program interruption," in *ARES*. IEEE, 2008, pp. 33–38.

[12] L. T. Clark *et al.*, "A dual mode redundant approach for microprocessor soft error hardness," *IEEE Transactions on Nuclear Science*, vol. 58, pp. 3018–3025, 2011.

[13] A. Rohani *et al.*, "An on-line soft error mitigation technique for control logic of VLIW processors," in *DFT*, 2012, pp. 85–91.

[14] E. Ochoa-Jiménez *et al.*, "Implementation of RSA signatures on GPU and CPU architectures," *IEEE Access*, vol. 8, pp. 9928–9941, 2020.

[15] R. L. Rivest *et al.*, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.

[16] C. Paar *et al.*, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[17] D. Boneh *et al.*, "On the importance of checking cryptographic protocols for faults," in *Eurocrypt*. Springer, 1997, pp. 37–51.

[18] F. Bao *et al.*, "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.

[19] T. C. Koylu *et al.*, "RNN-based detection of fault attacks on RSA," in *ISCAS*, 2020, pp. 1–5.

[20] M. Agoyan *et al.*, "How to flip a bit?" in *IOLTS*. IEEE, 2010, pp. 235–239.

[21] Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 361–372.

[22] J.-M. Schmidt *et al.*, "Optical fault attacks on aes: A threat in violet," in *FDTC*. IEEE, 2009, pp. 13–22.

[23] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities." *PNAS*, vol. 79, pp. 2554–2558, 1982.

[24] M. Demircigil *et al.*, "On a model of associative memory with huge storage capacity," *Journal of Statistical Physics*, vol. 168, pp. 288–299, 2017.

[25] H. Ramsauer *et al.*, "Hopfield networks is all you need," *arXiv preprint arXiv:2008.02217*, 2020.

[26] D. Krotov *et al.*, "Dense associative memory for pattern recognition," *Advances in neural information processing systems*, vol. 29, 2016.

[27] A. Waterman *et al.*, "The RISC-V instruction set manual-volume i: User-level isa-document version 2.2," *RISC-V Foundation (May 2017)*, 2017.

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

[29] "7 series FPGAs data sheet: Overview," Sep 2020. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds180_7Series_Overview.pdf

[30] "RI5CY: User manual," Apr 2019. [Online]. Available: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf

[31] A. Rhisheekesan *et al.*, "Control flow checking or not?(for soft errors)," *TECS*, vol. 18, pp. 1–25, 2019.