Microarchitecture-aware Fault Models: Experimental Evidence and Cross-Layer Inference Methodology

Ihab Alshaer*, Brice Colombier†, Christophe Deleuze*, Vincent Beroulle*, Paolo Maistri†
*Univ. Grenoble Alpes, Grenoble INP[1], LCIS, 26000 Valence, France
†Univ. Grenoble Alpes, CNRS, Grenoble INP[1], TIMA, 38000 Grenoble, France
{first.last}@univ-grenoble-alpes.fr

***Abstract*—Fault injection attacks are considered one of the major threats to cyber-physical systems. The increasing complexity of embedded microprocessors involves complicated behaviours in presence of such attacks. Realistic fault models are required to study code vulnerabilities and be able to protect digital systems from these attacks. However, inferring fault models using only limited observations of faulty microprocessors is difficult. In this article, we present experiments that show the difficulty of characterizing and modelling the fault injection effects. From there, we propose a complete approach for fault analysis to build proper fault models at different system levels, which will help in designing suitable countermeasures at reasonable cost.**

***Index Terms*—fault injection, software, RTL, hardware, microarchitecture.**

I. Introduction

With the widespread use of embedded systems in different areas of life, protecting these systems against malicious use becomes crucial. Digital systems contain sensitive information that can be effectively protected through cryptographic algorithms, often implemented in software on an embedded microprocessor. Such implementations, however, might be vulnerable to attacks that aim at extracting this sensitive information.

The protection task should even have a high priority, as the attack techniques and equipment are always improving. Fault injection is one of these attack techniques. In the context of hardware security, it can be defined as a powerful physical attack, possibly non-invasive, where the attacker has physical access to the device or its environment. The attacker will try to change the normal behaviour of the device during a program execution by injecting a fault, then observing the erroneous behaviour. The injection process can be done in different ways: exposing the device to radiations, laser beam, intense light or an electromagnetic (EM) pulse, inducing variations in the power supply, injecting a glitch in the clock signal, changing the environmental conditions such as the temperature, etc [1]. The resulting fault could reveal an interesting behaviour that could be further exploited as a vulnerability.

Securing components, such as microprocessors and microcontrollers, against fault attacks, requires a thorough understanding of faults: on the one hand, this means characterizing, studying, and analyzing the faults that could lead to exploitable code vulnerabilities. On the other hand, it also requires designing countermeasures at different levels, hardware and software, with an acceptable cost.

To build appropriate countermeasures, designers need realistic fault models that provide proper characterization of the fault effects. However, with the increasing complexity of microprocessors, fault effect characterization based on a single level of analysis, such as assembly level or Register-Transfer level (RTL), is difficult and limits the understanding of the fault. As a consequence, the fault model development becomes a complex task: the models are a high-level approximation, sometimes unrealistic, and the development and evaluation of the countermeasures are not optimized.

For the sake of designing countermeasures, several research studies have been conducted based only on a single level of fault characterization, such as Instruction-Set Architecture (ISA) level in [2], [3] or RTL [4]. However, because of the incomplete fault model, this could lead to either under-engineer or over-engineer the protections. In the former case, a security threat may still be present and hence exploitable; in the latter, this means unnecessarily adding cost and possibly decreasing performance.

Other studies tried to propose analysis by performing fault injection using more than one technique, in order to have a better overview of the observed faulty behaviours. Moro *et al.* [5], for example, carried out EM injection campaigns on a microcontroller and compared the observed behaviours with the results given by software simulation based on software fault models. Dureuil *et al.* [6] tried to generalize fault models as a result of performing laser and EM injections on RAMs and Flash memories of smart cards, they then simulated faults at the application level in order to provide a so-called "vulnerability rate" for such faults. A similar approach has been followed by Werner *et al.* [7]: the authors carried out laser fault injection along with software fault simulation. However, they focused mostly on performing multi-fault attacks rather than proposing new or more thorough fault models. In these works, the authors provide fault characterization at the software level, *i.e.* ISA and/or high level applications, benefiting from observed faulty behaviours produced by physical fault injections. Hence, they did not provide complete details of analyzing the fault at the microarchitectural level in order to show how the fault occurred internally. Therefore, they could not assess the realism of their fault models.

Funded by the French program Investissement d'avenir.
[1]Institute of Engineering Univ. Grenoble Alpes

Finally, in [8], Laurent *et al.*, suggested that fault injections using typical software fault models (such as instruction-skip and test-inversion) are no longer enough to characterize the observed faulty behaviours, in particular when targeting complex microprocessors that have a large number of internal elements, *i.e.* registers and combinational logic. In their work, they provided a comprehensive analysis to assess software fault models by performing RTL fault simulation on a RISC-V microprocessor [9]. However, physical fault injections were not performed to validate the realism of their proposed RTL fault models. Moreover, different microprocessors should be taken into account in order to generalize the assumptions of their work.

In this article, we present experiments that illustrate the difficulty of characterizing fault effects resulting from physical injections. In particular, we show that some of the obtained faulty behaviours are strongly related to the microarchitecture of the target. On the basis of this evidence, we propose a complete methodology to address such issues and bridge the gap between previous studies by providing a cross-layer analysis of code and microarchitectural vulnerabilities while performing fault injections at three distinct levels: hardware/-physical, RTL, and software levels. We aim at providing a full picture of fault characterization at multiple description levels, by taking into consideration microarchitectural specifications. This will help in assessing the realism of already existing fault models, eliminate unrealistic models, and possibly propose new ones. Such methodology will also help in designing countermeasures at an appropriate cost.

The rest of the article is organized as follows: Section II describes the experimental setup. The results and the analysis are presented in Section III. Section IV explains the proposed methodology and the article is concluded along with the perspectives in Section V.

II. Experimental Setup

Physical fault injection experiments have been performed in order to see if the obtained faulty behaviours can be easily characterized and if they are consistent when making modification to the target codes.

The following subsections present the fault injection technique we used, the target board, and the target program.

A. Clock Glitch Fault Injection

Applying perturbations to the clock signal that is fed to the microprocessor is an effective type of physical fault injection technique. During a normal execution, at every rising edge of the clock, an instruction is fetched by the microprocessor, while another instruction (previously fetched) is being decoded or executed in another stage of the pipeline. Fig. 1 shows a normal behaviour when having a regular clock signal.

When performing clock fault injection, a glitch is injected just before or after the rising edge of the clock. This glitch would appear as a new clock cycle for the microprocessor. Therefore, a new instruction is fetched and the instruction previously decoded is executed. However, as the glitch disrupts

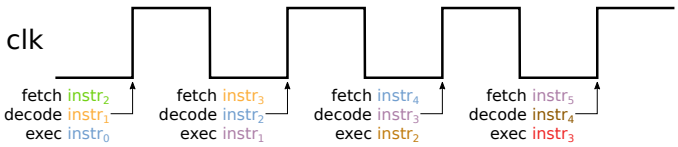

Fig. 1. Normal behaviour with a regular clock signal.

the regular behaviour of the clock signal, a timing violation will possibly occur, leading to various kinds of faulty behaviours.

When performing fault injection by clock glitch, the following parameters must be tuned:

- Delay: the time between the rising edge of the trigger signal used for synchronization and the rising edge of the targeted clock cycle;
- Shift: the time between the rising edge of the glitch and the rising edge of the targeted clock cycle.
- Width: the duration of the glitch itself.

Fig. 2 shows the glitch parameters with respect to a clock signal. It is worth mentioning that shift and width values should not be too large or too short. With too small values, the glitch will not be detected by the microprocessor, while too large values will allow the instructions to be executed normally without a fault.

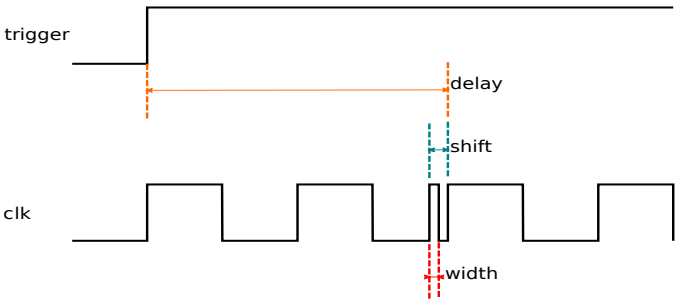

Fig. 2. Clock glitch parameters

B. Target Board

The board that is used for the experiments is a CW1173 ChipWhisperer-Lite [10]. It embeds a 32-bit microcontroller, which includes an ARM Cortex-M4 core [11]. The board is connected to a control PC through a USB cable. This board has a dedicated environment for side channel analysis, voltage and clock glitch of the target ARM core. We will leverage the clock glitch capabilities of this setup in the experiments.

The Cortex-M4 core supports the Thumb-2 instruction set [12]. It includes a pipeline with three stages: fetch, decode and execute. Up to two 16-bit instructions can be fetched at the same time. It also has a prefetch unit with a maximum size of six instructions.

C. Target Program

The injection is performed into inline assembly instructions within a C program. To ease the process of the injection, the program is divided into three parts as follows:

- Prologue: instructions for the initialization and the recording of the state before the injection happens.
- Target: instructions targeted by the fault injection as well as extra instructions that would allow observing any propagation effect.
- Epilogue: instructions for reading registers' state [R0-R12] and APSR[1] register (*i.e.*, Negative (N), Zero (Z), Carry (C) and Overflow (V) flags); the values are transferred through serial commands to the control PC.

Two series of NOP instructions are used to isolate the three parts. Three cases can occur as a result of the fault injection as follows:

- Crash: this class contains the cases where the fault injection causes a crash, a reset, or a failure when getting the final state from the board through the serial channel.
- Silent: this corresponds to the case when the outcome of the injection is identical to the golden state. We use the term golden state to refer to the outcome of a normal behaviour (*i.e.*, without any injection).
- Fault: when a successful fault has occurred and can be observed as a result of the fault injection.

In the injection campaign, we used specific instructions in the target part as shown in listing 1. This allows observing multiple things. Firstly, it shows if the resulting faulty behaviours are related to these instructions or not and hence, being able or not to characterize the faults. Secondly, it helps to understand if software characterization at the ISA level is sufficient to build realistic fault models based on the observations. For this reason, we used instructions that explicitly have effects on different architectural elements, such as the APSR flags. The glitch parameters were tuned to inject the fault in the first two instructions of the target part: CMP and BNE. The remaining instructions aim at observing possible propagation effects.

The campaign consists in repeating the clock glitch fault injection 10 000 times for the same shift, width and delay parameters. A single glitch is injected during each program execution. The registers R2 and R11 used in the experiment were initialized in the prologue to different values. Therefore, in a golden run, the zero flag remains clear, the branch is taken, and the instruction at line three is not executed.

```
1 CMP R2, R11 //r2-r11 then updates NZCV
2 BNE labelx //if (Z!=1): jump to labelx
3 ADD R10, R11, R2 //r10 = r11 + r2
4 labelx:
5 ADD R3, R11, R2 //r3 = r11 + r2
```

Listing 1. Target part in the target program

III. Results and Analysis

The results of the injection campaign are shown in Fig. 3. The x-axis presents the different observed faulty behaviours, while the y-axis shows their percentages over the successful faults *i.e.* without Crash and Silent cases. Complex faulty behaviours could appear as a combination of simpler models even if we only performed single fault injections. For example, the result of a single fault could be an instruction-skip and corruption of R0 at the same time.

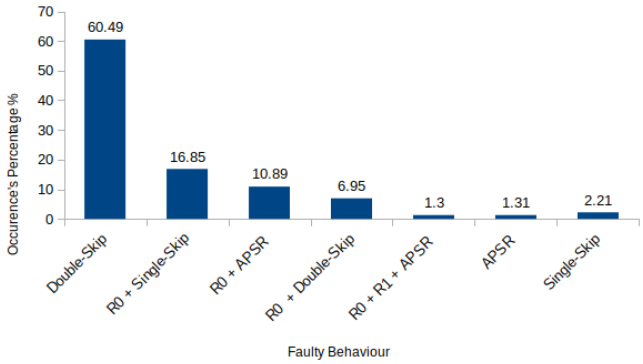

Fig. 3. Observed faults after the first campaign

During this campaign, Silent cases were only three over 10 000 and the same number was for the Crash category. Thus, 99.94 % of the injections were successful. The following faults have been observed:

- Skip: it can be either a single or a double-skip. In other words, either we skip the CMP instruction only, the BNE instruction only, or both. If APSR flags have not been updated, then we assume that the CMP instruction was skipped. If APSR flags have been updated correctly and the ADD instruction on line 3 is executed, then we assume that the BNE instruction was skipped. If APSR flags have not been updated and the ADD instruction on line 3 is executed, then we assume that both instructions were skipped.
- R0 corruption: this occurs when the value of R0 is different from its golden value. Among these corrupted values, we noticed the following: 0 (*i.e.*, the value of R0 becomes 0), right shift by 8 or 12 bits and other values with no obvious relation with the original value of R0.
- R1 Reset: R1 value becomes 0.
- APSR corruption: one or more of APSR flags have different values from the golden ones.
- Propagation effect on R10: it is caused by executing the ADD instruction on line 3. In this campaign, this instruction is always executed when having a successful fault. This can be explained as the consequence of two events. The first explanation is that the BNE instruction was skipped. The second explanation is that the Zero flag was corrupted. This leads to the branch not being taken as in a normal case, where the Zero flag is 0. Instead, as a result of the injection, the Zero flag was set to 1. These two cases could not be discriminated as both of them might even occur together.

A second campaign has been carried out with the same fault injection parameters (*i.e.*, shift, width and delay) and

[1]Application Program Status Register

initialization values but with a duplicated CMP instruction as shown in listing 2.

```
1 CMP R2, R11
2 CMP R2, R11
3 BNE labelx
4 ADD R10, R11, R2
5 labelx:
6 ADD R3, R11, R2
```

Listing 2. Target part in the second campaign

The second campaign has been performed in order to see if the faulty behaviours were consistent and to improve the understanding of the induced errors. In particular, its objective was to gain insight about the reason for the propagation effect on R10 as described above. In this campaign, there were no Silent cases while Crash occurred 7 times. Thus, 99.93 % of the injections were successful. This makes the two campaigns comparable in terms of population. The results are shown in Fig. 4. In addition to skip and APSR corruption, the following behaviours were observed:

- R0 corruption: again we observed occurrences of reset, right shift by 4, 8 or 12 bits and setting to other values, but with larger variability when compared to the previous campaign.
- R2 corruption: R2 has a large value that is different from the golden one.
- Propagation effect on R10: since we target only the first two instructions, this can not be caused by a skip or other perturbation of the BNE instruction. Therefore, this is necessarily caused by corruption of the Zero flag.
- Propagation effect on R3: as a result of the corrupted value in R2, R3 has a wrong value at the end, since it is the sum of R11 and R2.

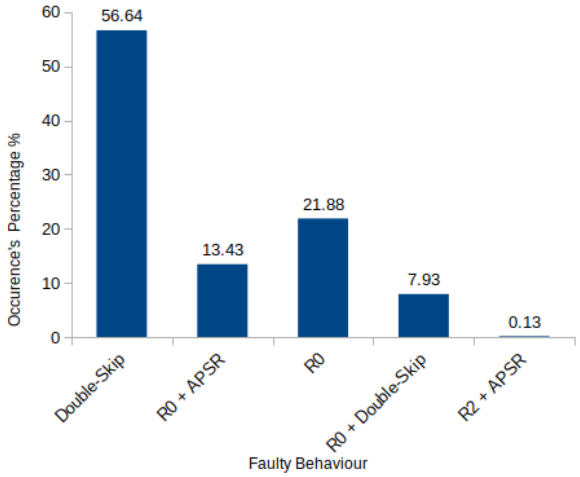

Fig. 4. Observed faults after the second campaign

These experimental results led to the following observations and questions:

- Small changes in the target program have large consequences on the observed faults: some faulty behaviours disappear such as the R1 Reset. New faults appear, for example, corruption in R2, which leads to a propagation effect on R3. And finally, different corrupted values are observed.
- Faulty behaviours may appear with different percentages. However, we have found that the occurrence probability of a specific behaviour can be improved by fine adjustments of the glitch parameters.
- Some registers that are not used in the program end up being corrupted as well: R0 and R1 in the first campaign, R0 in the second campaign. A question arises about what would be the proper fault model to account for this effect. In particular, such errors may have several causes: it might be related to the instruction opcode (*i.e.*, a fault during the instruction fetch) or to the execution stage of the pipeline.
- One might think that duplicating CMP will work as a countermeasure for APSR corruption, but it did not as the injection affects two instructions, which might be related to the microarchitectural feature of having the possibility to fetch two instructions at the same tame. Hence, the corruption of APSR might still be occurred as a result of either corruption in the second CMP or corruption in the first and skipping the second. However, we cannot ensure that a single-skip in one of the CMP instructions has occurred as executing one of them, either properly or improperly, will mask the single-skip effect. Thus, at this step we can only say that either double-skip or APSR corruption have occurred.
- The corruption of APSR flags can be due to several causes: a change in the registers values while executing CMP, an error that occurred when updating the APSR flags, a fault in a control signal related to the APSR flags... All these hypotheses cannot be answered without a better knowledge of the microarchitecture, which will help in having a suited fault model at the end.
- Most importantly, there is no explanation at this level for the corrupted values found in the registers: 0, shift, seemingly random values, etc. We believe that some of these values are related to the microarchitecture, which will affect how a corrupted instruction will be executed.

The aforementioned faults could be exploited as vulnerabilities in a security application. For example, an APSR corruption can lead to test-inversion where tests are considered very important in the control flow of critical applications.

To sum up, we saw how fault characterization is difficult based on a single level of analysis. These results show the difficulty of building consistent fault models that allow designers to predict the fault injection effects and design efficient and cost-effective countermeasures. Thus, additional research is necessary. In the next section, we propose a methodology that takes into consideration multiple levels of analysis by including software and RTL fault simulations as well as physical fault injections. This will help in explaining the observed points and answering the above-mentioned questions.

IV. Proposed Methodology

This section provides a full description of the proposed methodology to infer fault models that will help in designing fair cost hardware and software countermeasures. It deals with three different levels of understanding in order to provide a cross-layer fault analysis.

Fig. 5 depicts the proposed methodology. It is centered around a comparison between the obtained results that are stored in three databases (hardware, RTL and software databases) in order to make decisions about the consistency and applicability of RTL and software fault models. In other words, starting form the observations obtained at the lowest level of abstraction (*i.e.*, hardware level), it will be possible to optimize fault models at the RTL level, for example, by removing RTL faults that do no correspond to observable faulty outputs. Then, by using these RTL models, the models at software level will be optimized in a similar way, by adjusting them to not include behaviours that cannot be observed at RTL or hardware level. The following subsections explain each of these parts in more details.

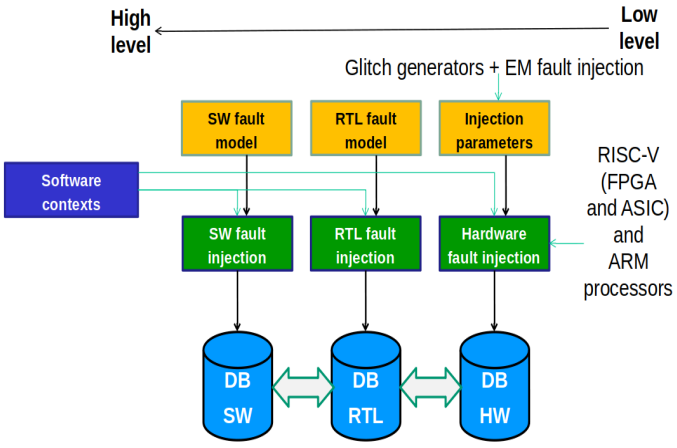

Fig. 5. Proposed Methodology.

A. Hardware Fault Injection

In this step, which is currently in progress as described in previous sections, the goal is to perform physical fault injections using a variety of injection techniques. Among these methods: EM fault injection, voltage and clock glitch injections using dedicated printed circuit boards and suitable generators. In each injection campaign, the following procedure will be applied:

- Define a wide range of software contexts as target programs for the injection process. Faults are going to be injected while executing these programs on one of the hardware targets, for instance, microcontrollers, Application Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA).
- Define the set of injection parameters. For example, in the case of clock glitch attacks, the range of values for the shift and the width of the glitch, as well as the delay, as described and explained in the previous sections. These parameters as well as the target board layout must be taken into account when describing the fault model.
- Get a snapshot of the fault injection: the registers and memory states will be read at the beginning and at the end of the program execution (using a serial communication link with the host PC or a debugger for example). Then it will be compared with the configuration of a golden run. The faulty behaviours will be stored in a database (HW DB in Fig. 5). This step will allow us to observe the relation between the observed faulty behaviours and the instructions in the target part. In other words, the aim is to assess if there is a direct relation (*i.e.*, the effect corresponds to the target instructions), an indirect relation (*i.e.*, the effect is a result of a propagation effect), or no relation at all, which may require further analysis.
- Thanks to the analysis of the observed faulty behaviours, a fault model inference process will be followed by generalizing the obtained faulty behaviours.

B. RTL Fault Injection

In order to understand what is exactly happening internally at the microarchitectural level and be able to know the origin of a fault, fault simulation campaigns are going to be performed on the RTL description of the microprocessor. This will help in characterizing further the hardware faulty behaviours.

With RTL fault simulation, it is possible to inject faults in a very precise manner into the microarchitecture. For instance, inter-stage pipeline registers, multiplexers and different arithmetic units that are involved in executing an instruction in the pipeline stages can be targeted. The injection will consist in forcing the corresponding signals, according to fault models such as single or multiple bit-flips, bit-sets and bit-resets.

As in the previous step, the resulting faulty behaviours will be stored in a dedicated database and then be compared with those obtained from the physical injections. This comparison will help in two aspects, as shown visually in Fig. 6. On the one hand, this aims at explaining at the hardware level the faulty behaviours obtained from physical injections, and hence, making the fault effect characterization easier. On the other hand, it also helps in validating and assessing the realism of the obtained RTL fault models. Hence, it provides a full overview to the hardware designer to build the required countermeasures.

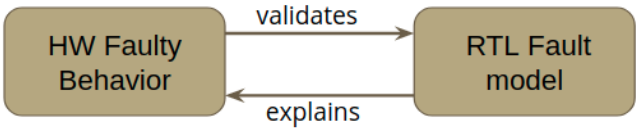

Fig. 6. Relation between hardware faulty behaviour and RTL fault model.

C. Software Fault Injection

Software faults will be injected into different target programs as mutants. This can be done by performing modi-

fication, deletion or addition of instructions in the original program. They may correspond to a large variety of faulty behaviours modeled at the ISA level. In other words, classic fault models such as instruction skip, instruction replacement, instruction corruption, register value corruption, test-inversion, etc, will be injected into the programs by modifying the instructions. In addition to that, other faulty behaviours must be generated using more complex fault models which take into consideration the modern design of some hardware blocks. This includes, for instance, forwarding and speculative execution. In this case, dedicated techniques shall be employed to model the advanced architectural characteristics and the related faults at the ISA level.

The expected faulty outputs will again be stored in a corresponding database. Then, a comparison process similar to the one mentioned earlier will take place between the RTL and software faulty results. In this step, an RTL model validates the consistency of a software model, whereas a software model will be usable to describe the occurrence and explain an RTL model at the application level, which makes the fault effect characterization at this level easier.

Once the links between the three levels are established and formalized, a software developer can design the most suitable countermeasures for a given context. For sure, countermeasures will be studied carefully at both levels: hardware and software. Therefore, the proper ones will be applied by taking into account their cost and their effect on the performance. Thus, if a countermeasure can be implemented at both hardware and software levels with comparable efficiency, only the software instance may be taken into account as software countermeasures are usually less expensive and easier to implement. Therefore, the "cross-layer" aspect can be extended later on to the design of countermeasures.

V. Conclusion and Perspectives

In this article, we presented the existing problems in analyzing and understanding fault attacks in complex microarchitecture. We highlighted this by providing experimental evidence of intrisically microarchitectural faults, using clock glitch as the fault injection technique. Then, we proposed a new methodology to provide a cross-layer analysis for characterizing faulty behaviours. This can be used to build realistic fault models at different levels such as RTL and software. Hence, this gives the possibility to design suited countermeasures at the most appropriate cost.

Although we have just performed simple clock glitch injection campaigns on simple software contexts, we were able to observe a variety of faulty behaviours. We believe that by applying other injection techniques on different software contexts, a larger set of faulty behaviours will be obtained, which will enrich the whole analysis. However, automating the analysis of the faulty behaviours and the comparison among the three different databases obtained at the different layers is necessary to move forward in this research direction.

Acknowledgment

This work has been supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) and the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-15-IDEX-02).

References

[1] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[2] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *Design, Automation and Test in Europe*, E. Macii, Ed., Grenoble, France, 2013, pp. 404–409.

[3] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner, "Qemu-based fault injection for a system-level analysis of software countermeasures against fault attacks," in *Euromicro Conference on Digital System Design*. Madeira, Portugal: IEEE Computer Society, 2015, pp. 530–533.

[4] S. Bergaoui, P. Vanhauwaert, and R. Leveugle:, "A new critical variable analysis in processor-based systems," *IEEE Transactions on Nuclear Science*, vol. 57, no. 4, pp. 1992–1999, 2010.

[5] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller," in *Workshop on Fault Diagnosis and Tolerance in Cryptography3*, W. Fischer and J. Schmidt, Eds. Los Alamitos, CA, USA: IEEE Computer Society, 2013, pp. 77–88.

[6] L. Dureuil, M. Potet, P. de Choudens, C. Dumas, and J. Clédière, "From code review to fault injection attacks: Filling the gap using fault model inference," in *International Conference on Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, N. Homma and M. Medwed, Eds., vol. 9514. Bochum, Germany: Springer, 2015, pp. 107–124.

[7] V. Werner, L. Maingault, and M. Potet, "An end-to-end approach for multi-fault attack vulnerability assessment," in *Workshop on Fault Detection and Tolerance in Cryptography*. Milan, Italy: IEEE, 2020, pp. 10–17.

[8] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, "Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor," *Microprocessors and Microsystems*, vol. 71, 2019.

[9] RISC-V Foundation, "The RISC-V instruction set manual," https://riscv.org/technical/specifications/, [Accessed: May 4, 2021].

[10] C. O'Flynn and Z. D. Chen, "Chipwhisperer: An open-source platform for hardware embedded security research," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*, ser. Lecture Notes in Computer Science, E. Prouff, Ed., vol. 8622. Paris, France: Springer, 2014, pp. 243–260.

[11] J. Yiu, *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*. Newnes, 2013.

[12] ARM Limited, "ARM architecture reference manual Thumb-2 supplement," https://developer.arm.com/documentation/ddi0308/d, [Accessed: May 4, 2021].