Defending the Citadel: Fault Injection Attacks against Dynamic Information Flow Tracking and Related Countermeasures

William Pensec, Francesco Regazzoni, Vianney Lapotre, Gogniat Guy

Defending the Citadel: Fault Injection Attacks against Dynamic Information Flow Tracking and Related Countermeasures

William PENSEC
Université Bretagne Sud
UMR 6285, Lab-STICC
Lorient, France
william.pensec@univ-ubs.fr

Francesco REGAZZONI
ALARI
University of Amsterdam - USI
Lugano, Switzerland
regazzoni@alari.ch

Vianney LAPÔTRE
Université Bretagne Sud
UMR 6285, Lab-STICC
Lorient, France
vianney.lapotre@univ-ubs.fr

Guy GOGNIAT
Université Bretagne Sud
UMR 6285, Lab-STICC
Lorient, France
guy.gogniat@univ-ubs.fr

***Abstract*—Embedded processors are key components of Internet of Things (IoT) devices of Cyber-Physical Systems (CPSs) that manipulate sensitive data. In order to mitigate software attacks, hardware-assisted Dynamic Information Flow Tracking (DIFT) has been integrated into embedded processors. Due to their proximity to attackers, IoT devices are also exposed to physical attacks such as Fault Injection Attacks (FIAs). In this paper, we protect DIFT from fault injection attacks by extending current DIFT support with fault detection and correction capabilities. To do so, we design, implement and evaluate two countermeasures based on parity bit or Hamming code to protect DIFT-related registers of RISC-V CPUs. Our experimental results, obtained using the D-RI5CY processor as a use case, show a 100% fault detection and fault correction when relying on a Hamming code-based protection and a low area overhead (10.6%) compared to the original design.**

***Index Terms*—Hardware security, RISC-V, DIFT, Fault Injections Attacks, Error-Correcting Code, Countermeasures**

I. Introduction

Internet-of-Things devices and Cyber-Physical Systems are used in numerous domains such as home automation, medical sensing, transports, critical infrastructures, smart security systems, etc. They manipulate sensitive data and, in some cases, they also include actuators to autonomously react to specific situations. Given the network connectivity that these devices have and their pervasive diffusion also in places possibly accessible by adversaries, these devices are vulnerable to both software and physical attacks. Literature demonstrates that combined software and physical attacks [1]–[4], are indeed a threat for these devices, giving the attacker the possibility to recover secret information or gain access to a device.

Dynamic Information Flow Tracking (DIFT) mitigates various software attacks such as buffer overflow, format string or malware by attaching and propagating tags to data containers at runtime [5]. Associated with a tag check security policy, it raises an alert when malicious behaviour is detected. Because of its suitability, multiple DIFT implementations have been studied in the literature: hardware, software, and hybrid DIFT [6]. However, as demonstrated for other processor's security measures (TrustZone [4] and RISC-V PMP [2] to mention a few), DIFT is open to physical attacks.

Addressing for the first time the problem of physical protection of DIFT, in this paper, we extend an in-core DIFT operating at the hardware level with error detection and protection capabilities. We consider as a use case the D-RI5CY processor [7] and we deal with an attacker able to inject faults in the DIFT-related registers. We implement and evaluate two hardware countermeasures, simple parity and Hamming Code, and we evaluate their suitability, robustness, and overhead by performing fault injection at a granularity of bit using a cycle-accurate simulator.

The rest of the paper is structured as follows. Section II presents related work. Section III introduces the D-RI5CY processor and explains the two protection mechanisms we are using in this paper. Section IV presents the design of the proposed countermeasures. Section V describes our experimental setup for fault injection simulations and details the methodology. Section VI presents the results of the different fault injections campaigns with and without countermeasures and compares them. Finally, Section VII concludes the work and draws some perspectives.

II. Related Work

DIFT monitors, at runtime, data flow of the application binary in order to detect software attacks or to prevent data leakage. In [8], the authors provide a comprehensive survey of the different Information Flow Tracking (IFT) solutions from static IFT to DIFT. They present both hardware and software IFT sub-categories. Information containers depend on which type of DIFT is used; these range from files to registers.

Hardware DIFT solutions can be grouped into two main categories: off-core and in-core. Off-core DIFT [9], [10] relies on a dedicated coprocessor to perform tag-related operations. This approach does not require internal processor modification and reduces the computation load on the main processor. However, the communication and synchronisation between the main processor and the coprocessor need to be carefully managed. [11] proposes a software/hardware method to protect the entire RISC-V-based SoC platform at runtime. They implemented a coprocessor with a dedicated bus and a DIFT-supported IP wrapper which does not change the architecture of the main core. Compared with the off-core approach, in-core DIFT does not require communication and synchronisation management, but it leads to invasive modifications to the processor. Tag-related operations are computed in parallel of data. For instance, works presented in [12] and [7] offer a flexible hardware/software approach, relying on an in-core hardware DIFT. These architectures allow for flexible and configurable security policies to protect against a wide range of attacks.

Despite these solutions providing an effective technique to address certain software attacks, their robustness and protection against physical attacks, such as fault injection attacks, are largely unexplored. FIA can be performed by disturbing the power supply or the clock, by using EM pulses or laser shots [13]. The impact of an injection varies depending on the type of FIA. For instance, laser-based injections provide the best spatial and temporal precision while the power supply or clock perturbation will affect the whole circuit, limiting the spatial precision.

Many studies have shown the vulnerabilities of critical systems against FIAs. [14] demonstrates that it is possible to recover computed secret data using FIA in hidden registers on the RISC-V Rocket processor. Electromagnetic fault injection (EMFI) attack can be used to recover an AES key by targeting the cache hierarchy and the MMU, as shown in [15]. Laser fault injections (LFI) can allow the replay of instructions [16], that can lead to the overwriting of an entire section of a program. [3] shows the use of glitch injections on the power supply to control the program counter (PC). Voltage glitches can also lead to glitch TrustZone mechanisms, as shown in [4]. Finally, authors of [2] have shown that one can combine side-channel attacks (SCA) and FIAs to bypass the PMP mechanism in a RISC-V processor. Despite this large amount of literature about FIA, to the best of our knowledge, the protection of DIFT against FIA has not been explored yet.

Different types of countermeasures can be used to protect security mechanisms against fault injection attacks. Numerous works in the literature have proposed countermeasures for cryptographic devices. [17] presents a survey of the different encryption algorithms (SNOW 3G, AES, RSA, Elliptic Curve Cryptography (ECC)) and presents the possible attacks on these algorithms. The authors present hardware countermeasures such as execution duplication, error detection code (EDC), nonlinear EDC, redundancy (time or space) and parity bits. When targeting processors, software countermeasures can also be implemented. [18] analyses 19 implementations of software countermeasure strategies and evaluates them in terms of performance and security against fault injection attacks into a microcontroller simulator based on an ARM Cortex-M3. Their results show that some simple countermeasures perform better than more complex ones due to their low memory or performance overhead. Moreover, combinations of countermeasures can lead to high fault coverage, while maintaining low resource overhead. However, software countermeasures usually do not take into account the processor micro-architecture (i.e., pipeline, hidden registers, optimisations) reducing the efficiency of the protections [19].

Since in-core DIFT mainly relies on a set of registers and logic elements that are not directly manipulated by the software, software countermeasures could be not practical to protect the entire DIFT. Moreover, [20] has shown that software countermeasures based on duplication and triplication can lead to significant overheads in terms of execution time (resp. 2x to 4x) and code size (resp. 4x to 14x per protected instruction). Because of this, in this work, we concentrate on hardware-supported countermeasures. Authors of [21] propose a solution based on Hamming Code and parity bits as signature generators for an AES cipher. The proposed solutions allow detecting faults while minimising the impact on timing performances, area and power consumption, making Hamming Code extremely suitable for our case. We selected to use Hamming Code because of this.

III. Background

A. D-RI5CY architecture

Figure 1 presents an overview of the D-RI5CY processor. DIFT-related modules are highlighted in red. These modules allow storing, propagating and checking tags during the execution of a sensitive application. 1-bit tags are stored parallel to the data they are associated with in the Data Memory and in the *Register File Tag*. The 1-bit tag associated with the PC is used to detect a malicious PC manipulation, for example during a return-oriented programming attack. The security policy is configured through two CSRs (*Configuration and Status Register*) named TPR (*Tag Propagation Register*) and TCR (*Tag Check Register*). The *Tag Update Logic* module is

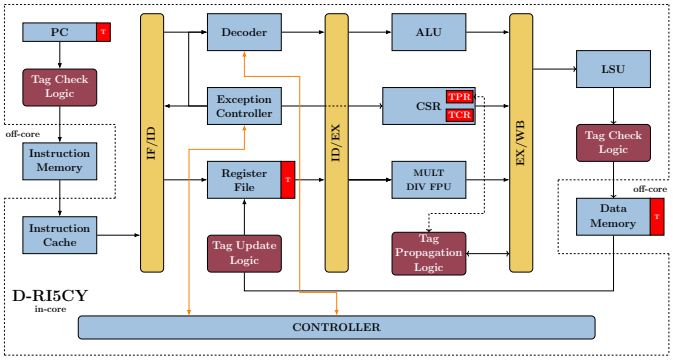

Fig. 1: D-RI5CY processor architecture overview

used to initialise or update the tag in the register file according to the tagged data. Then, when a tag is propagated in the pipeline, the *Tag Propagation Logic* module propagates tags according to the security policy defined in the TPR. Once a tag has been propagated and its data has been sent out of the pipeline, the *Tag Check Logic* modules check that it conforms to the security policy defined in the TCR. If not, an exception is raised. It is worth noting that the D-RI5CY designers have chosen to rely on the illegal instruction exception already implemented in the original RI5CY processor to manage the DIFT exceptions. This choice minimises the area overhead of the proposed solution.

Table I shows the TPR configuration for the security policy considered in this paper. Each instruction type has a user-configurable 2-bit tag propagation policy field (except for *Load/Store enable* which has a 3-bit tag) which is configured through a write instruction in the CSR. The tag propagation policy determines how the instruction result tag is generated according to the instruction operand tags. The configuration presented in Table I allows propagating tags from the source of a load/store and from the inputs for load/store, logic, shift, jump and arithmetic modes with a logical OR on both inputs.

Table II shows the TCR configuration for the considered security policy. Each instruction type has a user-configurable 3-bit tag control policy field (except for *Execute check*, *Branch check* and *Load/Store check* which have 1, 2 and 4-bit tag control policy fields respectively) which can be configured. The configuration presented in Table II checks the corruption of the PC address and the source and destination addresses of load/store operations. The tag verification policy determines whether the integrity of the system is corrupted based on the tags of the instruction's operands. [22] details how TPR (Table I) and TCR (Table II) configurations determine tag propagation and tag checking for the considered security policy.

B. Threat Model

Our threat model considers an attacker able to inject faults into DIFT-related registers, leading to bit-flips at any position of the targeted register. To bypass the DIFT mechanism, the main attacker's goal is to prevent an exception from being raised. To reach this objective, any DIFT-related register maintaining 1-bit tag value, driving the tag propagation or the tag update process or maintaining the security policy configuration can be targeted.

In [22], the authors study the impact of fault injections on the D-RI5CY processor. They show that the DIFT vulnerabilities concern mainly single bit-flip attacks. Indeed, DIFT computation relies on a 1-bit data path. Studies presented in [23] and [24] have shown that such precise single bit-flip attacks targeting registers can be performed using, for example, laser shots.

TABLE I: Tag Propagation Register configuration

	Load/Store Enable	Load/Store Mode	Logical Mode	Comparison Mode	Shift Mode	Jump Mode	Branch Mode	Arith Mode
Bit index	17 16 15	13 12	11 10	9 8	7 6	5 4	3 2	1 0
Policy	0 0 1	1 0	1 0	0 0	1 0	1 0	0 0	1 0

TABLE II: Tag Check Register configuration

	Execute Check	Load/Store Check	Logical Check	Comparison Check	Shift Check	Jump Check	Branch Check	Arith Check
Bit index	21	20 19 18 17	16 15 14	13 12 11	10 9 8	7 6 5	4 3	2 1 0
Policy	1	1 0 1 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0	0 0 0

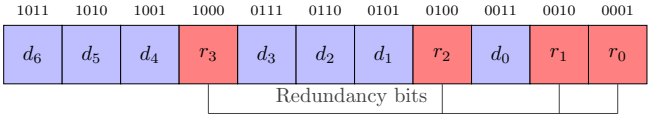

Fig. 2: Hamming codeword

Recent works such as [25] have shown that multi-spot laser fault injection setup can be used to improve attacks. Thus, we also consider an attacker able to inject a single bit-flip in two registers at two distinct clock cycles, with a minimum delay of one clock cycle. Consequently, in order to protect the D-RI5CY DIFT mechanism while minimising the area overhead, we choose to focus on lightweight code-based protections able to address single bit-flip faults.

C. Simple parity code

Error detection is often achieved through the use of parity codes, which involve adding an extra bit to the data bits for redundancy. Simple parity codes can detect single-bit errors. We selected simple parity code as a fault detection countermeasure because of its suitability and limited overhead. Furthermore, as highlighted by [22] DIFT vulnerabilities implemented in the D-RI5CY processor concern single bit-flip attacks which makes simple parity code an appropriate solution.

D. Hamming Code

Hamming Codes are a class of linear error-correcting codes invented by Richard W. Hamming [26] in 1950. The main use of these codes is to detect and correct errors. They are mostly used in digital communication and data storage systems as error control codes. Hamming Code can detect and correct single-bit errors or detect double-bits errors. This code is based on the principle of adding r redundancy bits to d data bits such that $2^r \geqslant d + r + 1$. For example, for an 8-bit word it needs 4 redundancy bits while for a 32-bit word, it needs only 6 redundancy bits. The redundancy bits are set to indexes equivalent to powers of 2 (1, 2, 4, 8, 16, ...) in the new concatenation between the word and the redundancy bits. By positioning the redundancy bits at the indexes of powers of two, it is then possible to correct an error if one is detected. Thus, for example, Hamming Code (11,7), 7-bit data ($d_0 - d_6$) and 4-bit of redundancy ($r_0 - r_3$), data bits and redundancy bits are placed according to figure 2.

Equation 1 shows how the redundancy bits are computed from the data bits.

$$\begin{aligned} r_0 &= d_0 \oplus d_1 \oplus d_3 \oplus d_4 \oplus d_6 \\ r_1 &= d_0 \oplus d_2 \oplus d_3 \oplus d_5 \oplus d_6 \\ r_2 &= d_1 \oplus d_2 \oplus d_3 \\ r_3 &= d_4 \oplus d_5 \oplus d_6 \end{aligned} \tag{1}$$

Hamming Code incurs a limited overhead, despite the correction capabilities. Because of this, we selected them to provide error correction capabilities to our DIFT.

IV. Fault Detection/Correction Implementation

The target architecture we used in this paper is the D-RI5CY DIFT. The D-RI5CY DIFT mechanism implements 55 registers, maintaining a total of 127 bits. Columns 2 and 3 of Table III detail the register distribution. Two 32-bit registers (TPR and TCR), a set of 32 1-bit registers building the tag register file, a 5-bit register maintaining the destination register file address and a set of 20 1-bit, 2-bit and 4-bit registers dedicated to control. Each of these registers needs to be protected against FIA.

The implementation cost of detection and correction codes highly depends on the bit-width of the protected value. For instance, considering Hamming Code, 2 redundancy bits have to be generated to protect a single bit while 6 redundancy bits are necessary for a 32-bit value. To minimise the overhead while maintaining a high level of protection, we have chosen to group registers to be protected into 5 groups. Table III presents the 5 groups we consider. 32-bit TCR and TPR configuration registers are individually protected. However, only up to 22 bits of these registers are actually used for the configuration of the DIFT mechanism. The tag register file maintaining 32 1-bit tags is protected as a single value (i.e., a unique parity bit or Hamming Code value is associated to the 32 1-bit registers). A 5-bit register storing the tag destination address is kept in a separate group to minimise the impact on the pipeline micro-architecture. Finally, 20 DIFT-related registers dedicated to tag propagation and control are grouped to form a 26-bit value to be protected. Table III shows that to detect a single bit-flip in the DIFT-related registers using a simple parity code, 5 extra 1-bit registers need to be implemented while to detect and correct such faults using Hamming Code, only three 5-bit, one 6-bit and one 4-bit extra registers are necessary.

Figure 3 presents the proposed code-based protection scheme for independent registers (control signals are not reported, preserving the figure clarity). This solution has been adopted for groups 1, 2, 4 and 5 of Table III. In this approach, a set of registers is associated to a unique parity bit when considering simple parity code or by redundancy bits when using Hamming Code. This information is generated based on the register inputs by an *Encoder* and stored in a dedicated extra register. A *Decoder* allows the detection of an error caused by a fault injection, leading to a bit-flip in one of the protected registers. When implementing Hamming Code-based protection, the *Decoder* produces corrected outputs presented in dashed arrows in Figure 3. In the proposed approach, these results are 1) propagated to the rest of the design to ensure its correct behaviour and 2) multiplexed with protected registers inputs in order to correct the

TABLE III: DIFT-related protected registers

	Protected register	Number of protected bits	Number of parity bits for Simple Parity	Number of redundancy bits for Hamming Code
Group 1	TCR	22	1	5
Group 2	TPR	22	1	5
Group 3	Register File (Tag)	32	1	6
Group 4	Tag destination address 16×1-bit registers	5	1	4
Group 5	3×2-bit registers 1×4-bit register	26	1	5
Total		107	5	25

stored value when necessary (i.e., when the faulty register is not written with a fresh input).

Figure 4 depicts the proposed code-based protection scheme for the tag register file (group 3 of Table III) (as before, control signals are not reported for clarity). In the proposed scheme, a set of 32 1-bit registers is associated to 1 parity bit with the simple parity protection or to 6 redundancy bits when using Hamming Code. As in the previous case, the *Decoder* allows the detection of an error due to a bit-flip fault in one of the registers. With Hamming Code protection, the *Decoder* produces corrected outputs (dashed arrows) which are propagated to the tag register outputs. If a fault is detected, the corrected output is forwarded to the tag register interface. As soon as one of the two input ports is available, this corrected value is stored in the faulty register to correct the detected fault. A fresh input value has priority on the corrected value to ensure the data flow correctness. It is worth noting that to minimise the impact on the original D-RI5CY tag register file design, we have chosen to rely on the existing 2 input ports interfaces instead of adding a third input port dedicated to correction.

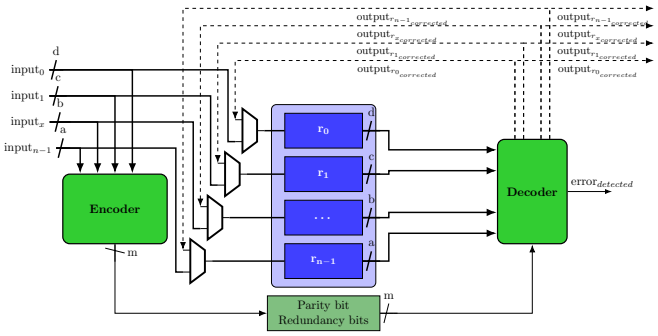

Fig. 3: Proposed scheme for code-based protections of a set of independent registers

V. Evaluation setup

In this section, we present our methodology for fault injection and the simulation campaign performed with the 2 use cases selected to stimulate the DIFT-related hardware modules.

We created a TCL script generator for Siemens Questasim to simulate the core and DIFT and to automatically handle the simulations and classifications of the simulation results.

Faults are injected into all 55 DIFT-related registers at cycle-accurate and bit-accurate levels. Moreover, when considering a protected design, countermeasure-related registers are also targeted. Each simulation logs information such as stop cycle, end-status, data in the registers file and associated tags, instruction register, etc. This data set is used to classify the end-of-simulation status. As in [22],

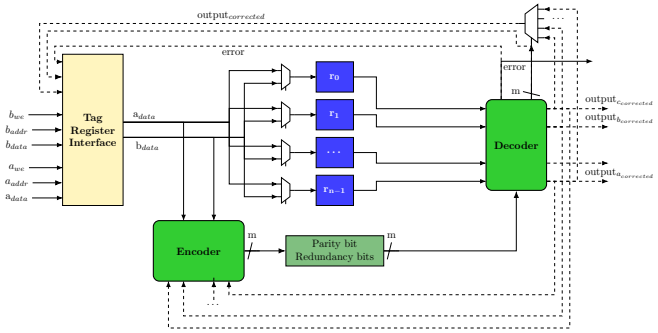

Fig. 4: Proposed scheme for code-based protections of the D-RI5CY tag register file

to stimulate DIFT-related hardware modules of the D-RI5CY DIFT mechanism when considering FIA, we have implemented two use cases: *buffer overflow* and *format string attack* relying on security policy from Tables I and II. The first use case enables the DIFT protection on the program counter (PC) when the attacker tries to modify the return address register. The second use case activates the DIFT protection in order to monitor load and store operations. The next subsections describe these use cases.

A. First use case: buffer overflow attack

The first use case involves a buffer overflow leading to a Return-Oriented Programming (ROP) attack[1] and subsequent shell code execution. In this case, the attacker leverages a buffer overflow to access the return address (ra) register. The DIFT mechanism comes into play as the tag associated with the manipulated buffer data overwrites the ra register's tag. Since the user manipulates the buffer data, it's tagged as *untrusted* (tag value = 1). During the return from the called function, the corrupted ra register is loaded into *PC* through a *jalr* instruction. This hijacks the execution flow, allowing the first shell code instruction to be fetched. This attack scenario sheds light on how DIFT behaves when monitoring the PC tag. The attack window for this use case represents 6 clock cycles where we fault the DIFT-related part of the core.

B. Second use case: format string attack

The second use case involves a format string attack[2], where the attacker exploits a vulnerability to overwrite a function's return address and execute shell code. In this attack, the `printf()` function from the C library is exploited using `%u` and `%n` formats (see Chapter 12,

[1]https://github.com/sld-columbia/riscv-dift/blob/master/pulpino_apps_dift/wilander_testbed/

[2]https://github.com/sld-columbia/riscv-dift/tree/master/pulpino_apps_dift/wu-ftpd

TABLE IV: Logical fault injection simulation campaigns results for single bit-flip in one register at a given clock cycle

		Crash	Silent	Delay	Detection	Detection & Correction	Success	Total
Buffer overflow	No protection	0	738	12	—	—	12 (1.57%)	762
	Simple parity code	0	0	0	792	—	0	792
	Hamming Code	0	0	0	—	912	0	912
Format String	No protection	0	946	41	—	—	29 (2.85%)	1,016
	Simple parity code	0	0	0	1,056	—	0	1,056
	Hamming Code	0	0	0	—	1,216	0	1,216

TABLE V: Logical fault injection simulation campaigns results for single bit-flip in two registers at two clock cycles

		Crash	Silent	Delay	Detection	Detection & Correction	Success	Total
Buffer overflow	No protection	0	238,633	1,143	—	—	2,159 (0.89%)	241,935
	Simple parity code	0	0	0	261,360	—	0	261,360
	Hamming Code	0	0	0	—	346,560	0	346,560
Format String	No protection	0	429,260	12,192	—	—	10,160 (2.25%)	451,612
	Simple parity code	0	0	0	487,872	—	0	487,872
	Hamming Code	0	0	0	—	646,912	0	646,912

Section 12.14.3 in [27] for detailed information). `%u` prints unsigned integer characters, while `%n` stores the number of printed characters in memory. The value used for `%n`, referred to as '`a`', is user-defined and tagged as untrusted for DIFT computations. The vulnerable statement is: *printf("%224u%n%35u%n%253u%n%n", 1, (int*) (a-4), 1,(int*) (a-3), 1, (int*) (a-2), (int*) (a-1))*. This leads to writing specific values at various memory addresses, with the attacker's goal being to overwrite the return address with 0x3e0 to trigger a shell code execution. However, security policy prohibits the use of untrusted variables as store addresses. Since variable '`a`' is untrusted, the DIFT protection raises an exception when trying to store a value at memory address '`(a-4)`'. This use case has been selected to activate the load/store modes of the DIFT policy. The attack window for this use case represents 8 clock cycles where we fault the DIFT-related part of the core.

VI. Experimental results

This section presents logical fault injection simulation results considering two fault models: single bit-flip in one register at a given clock cycle and single bit-flip in two registers at two clock cycles. Regarding the first fault model, Table IV shows the results obtained for the 2 considered use cases with and without protections. For protected implementations, faults are injected into both DIFT-related and protection-related registers. It is worth noting that we never get any crashes since we target the DIFT-related registers only. These registers do not impact the control or instruction flow executed by the processor. This table shows that without any protection 12 fault injections among 762 (1.57%) lead to a successful attack (i.e., bypassing the DIFT mechanism) for the buffer overflow use case while we observe 29 successes among 1,016 (2.85%) for the format string use case. Table V presents the results obtained considering the single bit-flip in two registers at two clock cycles fault model. It shows that without any protection 2,159 fault injections among 241,935 (0.89%) lead to a successful attack for the buffer overflow use case while we observe 10,160 successes among 451,612 (2.25%) for the format string use case.

In Table IV and V, column "*Silent*" means that the fault did not impact the current simulation behaviour. Even if we can consider that these numbers of successes are low compared to the total number of simulations, it should be noted that an attacker only needs to succeed once to gain access to the system. If we consider, as raised by [24], that the attacks are reproducible, if the attacker succeeds 1 time in one place he can start again as many times as necessary. It is therefore crucial to protect the system even if only a small number of attacks succeed. Despite these cases do not lead to a DIFT bypass, we can note that fault injections can cause a delay in the exception signal. It is worth noting that the number of simulations differs for each scenario because parameters such as the attack window and the number of target registers depend on the use case and the implemented protection (more registers for Hamming Code than for parity bit protection, as shown in Table III).

TABLE VI: FPGA implementation results

Protection	Number of LUT	Number of FF	Maximum frequency
D-RI5CY	6714	2457	47.6 MHz
Simple parity	7023 (4.6%)	2491 (1.4%)	47.1 MHz (-1.05%)
Hamming Code	7419 (10.5%)	2640 (7.4%)	46.5 MHz (-2.31%)

Using simple parity code allows detecting 100% of the injected faults. However, these faults are not corrected, and the program execution is halted. Results presented in Tables IV and V show that relying on Hamming Code provides an efficient solution to detect and correct all injected faults. In this case, the correct program execution is guaranteed, and the software attack is detected by the DIFT mechanism.

Table VI shows implementation results targeting the Xilinx Zynq-7000 of the Zedboard development board. Synthesis and implementation is performed using Vivado 2015.1 for 1) the original processor without any protection against FIA, 2) a processor version implementing simple parity code to protect the DIFT-related registers and 3) a processor version implementing Hamming Code for fault detection and correction. Results presented in Table VI allow a comparison in terms of timing performances and area overheads. Regarding LUT, protection leads to an overhead of 4.6% for simple parity protection and 10.5% for Hamming Code protection compared to the original design. The number of flip-flops grows by 1.4% for the simple parity protection and up to 7.4% for Hamming Code. Regarding maximum frequency, we observe that the proposed protections have a negligible impact on timing performances for the targeted FPGA.

Experiments results presented in Tables IV, V and Table VI show

that protecting DIFT-related registers of the D-RI5CY processor using Hamming Code to detect and correct all possible single bit-flip faults leads to an area overhead of only 6% compared to a simple parity code used for fault detection only. These results claim in favour of the adoption of Hamming Code-based protection scheme in such a small processor.

VII. Conclusion

This paper proposes to protect an in-core DIFT mechanism against fault injection attacks. The proposed approach relies on detection and correction codes to protect DIFT-related registers. Two protection schemes have been proposed. The first one provides a solution to protect a set of independent registers, while the second one is dedicated to the protection of the tag register file. Experimentation results show that the proposed approach allows the detection and correction of any single bit-flip in DIFT-related registers. Moreover, the proposed FPGA implementation of the proposed solutions for the D-RI5CY shows a maximum area overhead of 10.5% and no impact on timing performances.

In the future, we plan to extend this work by considering more complex fault models. This includes multiple injection capabilities and multi-bits fault models. We will study the impact of such fault models on the proposed protection scheme and propose original protection schemes when necessary. Furthermore, we plan to study the impact of register groups composition on proposed protection efficiency. Finally, we would like to perform actual fault injection campaigns to complete our work.

Acknowledgment

This work was supported by the *Collège Doctoral de Bretagne (France)*, the *GDR ISIS* research group, and the *Région Bretagne*.

References

[1] R. Schilling, P. Nasahl, M. Unterguggenberger, and S. Mangard, "SFP: Providing System Call Flow Protection against Software and Fault Attacks," in *Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '22, Association for Computing Machinery, 2022.

[2] S. Nashimoto, D. Suzuki, R. Ueno, and N. Homma, "Bypassing Isolated Execution on RISC-V using Side-Channel-Assisted Fault-Injection and Its Countermeasure," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021. DOI: 10.46586/tches.v2022.i1.28-68.

[3] N. Timmers, A. Spruyt, and M. Witteman, "Controlling PC on ARM Using Fault Injection," in *Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016. DOI: 10.1109/FDTC.2016.18.

[4] M. Saß, R. Mitev, and A.-R. Sadeghi, "Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M," in *USENIX security symposium*, 2023.

[5] W. Hu, A. Ardeshiricham, and R. Kastner, "Hardware Information Flow Tracking," *ACM Computing Surveys*, 2021. DOI: 10.1145/3447867.

[6] K. Chen, X. Guo, Q. Deng, and Y. Jin, "Dynamic Information Flow Tracking: Taxonomy, Challenges, and Opportunities," *Micromachines*, 2021. DOI: 10.3390/mi12080898.

[7] C. Palmiero, G. Di Guglielmo, L. Lavagno, and L. P. Carloni, "Design and Implementation of a Dynamic Information Flow Tracking Architecture to Secure a RISC-V Core for IoT Applications," in *High Performance Extreme Computing*, 2018. DOI: 10.1109/HPEC.2018.8547578.

[8] C. Brant, P. Shrestha, B. Mixon-Baca, *et al.*, "Challenges and Opportunities for Practical and Effective Dynamic Information Flow Tracking," *ACM Computing Surveys (CSUR)*, 2021. DOI: 10.1145/3483790.

[9] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling Dynamic Information Flow Tracking with a Dedicated Coprocessor," in *International Conference on Dependable Systems & Networks*, IEEE, 2009. DOI: 10.1109/DSN.2009.5270347.

[10] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, V. Lapôtre, and G. Gogniat, "Armhex: A hardware extension for dift on arm-based socs," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017. DOI: 10.23919/FPL.2017.8056767.

[11] K. Chen, L. Sun, and Q. Deng, "Hardware and Software Co-verification from Security Perspective in SoC Platforms," *Journal of Systems Architecture*, 2022. DOI: 10.1016/j.sysarc.2021.102355.

[12] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A Flexible Information Flow Architecture for Software Security," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Association for Computing Machinery, 2007. DOI: 10.1145/1250662.1250722.

[13] D. Karaklajić, J.-M. Schmidt, and I. Verbauwhede, "Hardware Designer's Guide to Fault Attacks," *IEEE Transactions on Very Large Scale Integration Systems*, 2013. DOI: 10.1109/TVLSI.2012.2231707.

[14] J. Laurent, V. Beroulle, C. Deleuze, and F. Pebay-Peyroula, "Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures," in *Design, Automation & Test in Europe Conference (DATE)*, 2019. DOI: 10.23919/DATE.2019.8715158.

[15] T. Trouchkine, S. K. K. Bukasa, M. Escouteloup, R. Lashermes, and G. Bouffard, "Electromagnetic Fault Injection Against a Complex CPU, toward new Micro-architectural Fault Models," *Journal of Cryptographic Engineering*, 2021. DOI: 10.1007/s13389-021-00259-6.

[16] V. Khuat, J.-M. Dutertre, and J.-L. Danger, "Analysis of a Laser-induced Instructions Replay Fault Model in a 32-bit Microcontroller," in *Digital System Design (DSD)*, 2021. DOI: 10.1109/DSD53832.2021.00061.

[17] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, 2012. DOI: 10.1109/JPROC.2012.2188769.

[18] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013. DOI: 10.7873/DATE.2013.092.

[19] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software fault resistance is futile: Effective single-glitch attacks," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016. DOI: 10.1109/FDTC.2016.21.

[20] 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*, Association for Computing Machinery, 2010. DOI: 10.1145/1873548.1873555.

[21] F. E. Potestad-Ordóñez, E. Tena-Sánchez, A. J. Acosta-Jiménez, C. J. Jiménez-Fernández, and R. Chaves, "Design and evaluation of countermeasures against fault injection attacks and power side-channel leakage exploration for aes block cipher," *IEEE Access*, 2022. DOI: 10.1109/ACCESS.2022.3183764.

[22] W. Pensec, V. Lapôtre, and G. Gogniat, "Another break in the wall: Harnessing fault injection attacks to penetrate software fortresses," in *Proceedings of the First International Workshop on Security and Privacy of Sensing Systems*, Association for Computing Machinery, 2023. DOI: 10.1145/3628356.3630116.

[23] F. Courbon, P. Loubet-Moundi, J. J. Fournier, and A. Tria, "Adjusting laser injections for fully controlled faults," in *Constructive Side-Channel Analysis and Secure Design*, Springer, 2014. DOI: 10.1007/978-3-319-10175-0_16.

[24] L. Zussa, J.-M. Dutertre, J. Clédière, B. Robisson, and A. Tria, "Investigation of timing constraints violation as a fault injection means," in *27th Conference on Design of Circuits and Integrated Systems (DCIS)*, Nov. 2012. [Online]. Available: https://hal-emse.ccsd.cnrs.fr/emse-00742652.

[25] B. Colombier, P. Grandamme, J. Vernay, *et al.*, "Multi-spot laser fault injection setup: New possibilities for fault injection attacks," in *Smart Card Research and Advanced Applications*, Cham: Springer International Publishing, 2022, pp. 151–166, ISBN: 978-3-030-97348-3.

[26] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, 1950. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[27] S. Loosemore, R. M. Stallman, R. McGrath, A. Oram, and U. Drepper. "The GNU C Library Reference Manual." (2023), [Online]. Available: https://www.gnu.org/s/libc/manual/pdf/libc.pdf.