Another Break in the Wall: Harnessing Fault Injection Attacks to Penetrate Software Fortresses

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

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

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

ABSTRACT

Internet of Things (IoT) devices manipulate sensitive data leading to strict security needs. They face both software and physical attacks due to their network connectivity and their proximity to attackers. These devices are usually built around low-cost and low-power processors. In this paper, we study the impact of Fault Injection Attacks (FIA) on Dynamic Information Flow Tracking (DIFT) mechanism of the D-RI5CY processor. Our results highlight the high sensitivity of this protection mechanism to multiple fault types at multiple spatial and temporal locations. Out of 3318 simulations, we achieved 74 successes (2.23%), mainly due to bit-flips.

KEYWORDS

Hardware security, RISC-V, DIFT, Fault Injections Attacks

ACM Reference Format:
William PENSEC, Vianney LAPÔTRE, and Guy GOGNIAT. 2023. Another Break in the Wall: Harnessing Fault Injection Attacks to Penetrate Software Fortresses. In *International Workshop on Security and Privacy of Sensing Systems (Sensors S&P '23), November 12–17, 2023, Istanbul, Turkiye.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3628356.3630116

1 INTRODUCTION

Internet-of-Things devices are used in numerous safety-critical domains as medical sensors, automotive, smart security systems, etc. Given their network connectivity and physical proximity to users, these devices are vulnerable to both software and physical attacks. A number of studies have already explored combined software and physical attacks [15, 11, 17, 14], with the aim of recovering information or gaining access to a device. Dynamic Information Flow Tracking (DIFT) detects various software attacks such as buffer overflow, SQL injections or malware by attaching and propagating tags to data containers at runtime [5, 2]. Associated with a tag check security policy, it raises an alert when malicious behaviour is detected.

In this paper, we consider the in-core DIFT implemented in the D-RI5CY processor [12]. We study the impact of Fault Injection Attacks (FIA) on the effectiveness of the D-RI5CY DIFT mechanism. We perform fault injection simulations to highlight the sensitivity of the D-RI5CY DIFT and determine the DIFT-related registers to be protected to counter fault injection attacks.

The main contributions of this work are:

- A fault simulation campaign to identify vulnerable DIFT-related registers in the D-RI5CY processor;
- An in-depth analysis of faults propagation along the DIFT hardware modules.

The rest of the paper is structured as follows. Section 2 presents related works. Section 3 introduces the D-RI5CY processor and the considered threat model. Section 4 presents our experimental setup for fault injection simulations and details the obtained results. Section 5 provides an in-depth analysis of the simulation results leading to successful attacks. Finally, Section 7 concludes the work and draws some perspectives.

2 RELATED WORKS

In [1], authors provide a comprehensive survey of the different types of Information Flow Tracking (IFT) solutions from static IFT to DIFT. They present both hardware and software IFT solutions. Hardware DIFT solutions can be grouped into two main categories: off-core and in-core. Off-core DIFT [6, 16, 3] relies on a dedicated co-processor 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 synchronization between the main processor and the co-processor need to be carefully managed. In-core DIFT leads to internal modifications of the processor. Tag-related operations are spread over the pipeline stages and are computed in parallel with the data treatments. Compared with the off-core approach, it does not require specific communication and synchronization management. However, significant invasive changes to the processor are required. For instance, works presented in [4] and [12] 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 tackle software attacks, they have not been evaluated against physical attacks such as fault injection attacks.

FIA can be performed by disturbing the power supply or the clock, by using EM pulses or laser shots [7]. Many studies have shown the vulnerabilities of critical systems against FIAs. [9] 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 [18]. Laser

Sensors S&P '23, November 12–17, 2023, Istanbul, Turkiye

ACM ISBN 979-8-4007-0440-6/23/11…$15.00
https://doi.org/10.1145/3628356.3630116

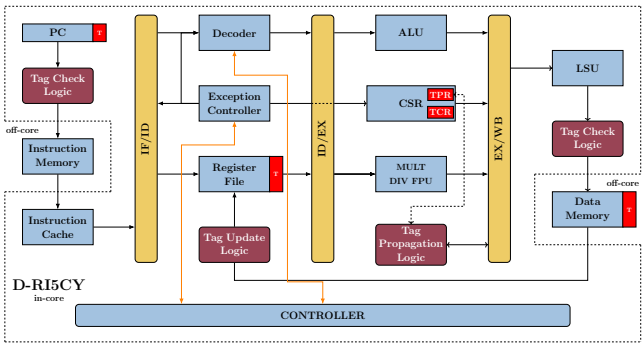

Figure 1: D-RI5CY processor architecture overview. DIFT-related modules are highlighted in red.

fault injections (LFI) can allow the replay of instructions [8]. [17] shows the use of glitch injections on the power supply to control the program counter (PC). Voltage glitches can also lead to glitch trust-zone mechanisms as shown in [14]. Finally, authors of [11] have shown that one can combine side-channel attacks (SCA) and FIAs to bypass the Physical Memory Protection (PMP) mechanism in a RISC-V processor.

To the best of our knowledge, in-core DIFT mechanisms have not been yet evaluated against FIAs. This paper presents the first security evaluation of a DIFT mechanism against physical attacks.

3 BACKGROUND

3.1 D-RI5CY

Figure 1 presents an overview of the D-RI5CY processor. DIFT-related modules are highlighted in red. These modules allow to store, propagate and check tags during the execution of a sensitive application. Tags are stored parallel to the data they are associated with in the Data Memory using 4 bits and in the *Register File Tag* using 1 bit. 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 Control/Status Registers (CSR) named TPR (*Tag Propagation Register*) and TCR (*Tag Check Register*).

The *Tag Update Logic* module is used to initialize 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 minimizes the area overhead of the proposed solution.

Table 1 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. For 2-bit fields, value '00' disables the tag propagation and the output tag keeps its previous value, value '01' stands for a logic AND on the 2 operand tags, value '10' stands for a logic OR on the 2 operand tags and value '11' sets the output tag to zero.

The *Load/Store Enable* field provides a finer-granularity rule to enable/disable the input operands before applying the propagation rule specified in the *Load/Store Mode* field. This extra tag propagation policy is defined through 3 bits. These bits allow to enable the source, source-address, and destination-address tags, respectively.

Table 2 shows the TCR configuration considered in this paper. 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 through the same instruction used for the TPR. The tag control policy determines whether or not the integrity of the system is corrupted based on the tags of the instruction's operands. The default 3-bit field should be read as follows: the right bit corresponds to input operand 1, the middle bit corresponds to input operand 2 and the left bit corresponds to the output tag of the operation. For each bit set, the corresponding tag is checked to determine whether an exception must be raised. The *Execute check* field is used to check the integrity of the PC. The *Branch check* field is used to check both inputs during branch instruction (*beq, blt, ...*). The right bit is used for input operand 1 and the left bit is used for input operand 2. Finally, the *Load/Store check* field is used to enable/disable source or destination tags checking during a *load* or *store* instruction. These bits enable or disable the checking of the source tag, source address tag, destination tag and destination address tag.

To illustrate the use of TCR and TPR registers, let's consider the detection of buffer overflow attacks leading to an ROP attack by overwriting a function return address. We assume that buffer data tags are set to 1 (i.e., *untrusted*) since the buffer is manipulated by the user. To detect this kind of attack, it is necessary to ensure the PC integrity by prohibiting the use of untrusted data for this register (i.e., *Execute Check* field of TCR set to 1). Regarding tag propagation configuration, load and store input operand tags must be propagated to output. Thus, the TPR register *Load/Store Mode* field should be set to value 10 (i.e., destination tag = source tag) and the *Load/Store Enable* field must be set to 001 (i.e., Source tag enabled).

3.2 Threat Model

We consider an attacker able to inject faults into DIFT-related registers leading to set to 0, set to 1 and bit-flips at any position of the targeted register. Indeed, [19] shows the ability of an attacker using techniques such as voltage/clock glitch to induce single-bit faults due to setup/hold time violations in flip-flops. Authors demonstrate that such attacks can be performed with a success rate beyond 90% and a reproducibility rate of 100%.

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 tag value, driving the tag propagation or the tag update process or maintaining the security policy configuration can be targeted.

Table 1: 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 2: 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

4 VULNERABILITY ASSESSMENT

Circuit sensitivity against FIA can be evaluated through actual physical attacks, simulations or using formal methods approaches. Physical attacks campaign confirms the sensitivity of a circuit to a specific technique of fault injection but it does not provide detailed information regarding the effect of the fault on the micro-architecture since the internal design registers cannot be monitored during the campaign. Formal methods approaches such as [13] allow the analysis of a circuit design in order to detect sensitive logic or sequential hardware elements. However, this type of tool usually suffers from restrictions limiting its actual usage on a processor. In particular, the circuit structure it can analyze. Thus, in this paper, a logical fault injection simulations campaign is performed to evaluate the sensitivity of the DIFT mechanism against FIA. In this campaign, we focus on both tag propagation and tag-checking computations. Faults are injected in the Hardware Description Language (HDL) code at cycle-accurate and bit-accurate levels. In this section, we present our methodology for fault injection and the simulation campaigns considering 2 use cases implemented to stimulate the DIFT-related hardware modules.

We developed a TCL script generator for Siemens Questasim 10.6e. It considers a set of input parameters such as fault types, target registers, target codes, the maximum number of cycles to be simulated and an attack window (temporal window in which a fault can be injected in each simulated clock cycle). Multiple attack windows can be defined for a simulation campaign. A non-faulty execution called *reference*, is first scheduled to provide a reference processor state to be compared with the ones from faulty simulations. We class the end status of each simulation into 4 categories: 1) Crash: the reference cycle count is exceeded, 2) Nothing Significant To Report (NSTR): no difference with the reference simulation, 3) Delay: illegal instruction exception time is delayed compared with the reference, 4) Success: the DIFT mechanism is bypassed.

Faults are injected into the 55 DIFT-related registers (a total of 127 bits) at cycle-accurate and bit-accurate levels. This fault injection campaign represents a total of 3318 faulted simulations considering the 2 use cases (use case 1: 1422 simulations / use case 2: 1896 simulations). The number of simulations is computed as follows: $(attackWindow * numberRegisters * 2) + (attackWindow * numberBits)$. The multiplication by 2 is due to the "set to 0" and "set to 1" threats, which only target the entire register. For example, for the first use case, the attack window is 6 cycles, 55 targeted registers and 127 bits. We then have: $(6 * 55 * 2) + (6 * 127) = 660 + 762 = 1422\ simulations$.

To identify vulnerabilities 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 1 and 2. The first use case enables the DIFT protection in order to monitor the program counter (PC). The second use case relies on the DIFT protection to monitor load and store operations. It is worth noting that monitoring arithmetic, logical, comparison, shift, jump or branch operations stimulates identical DIFT-related hardware modules. Since load and store operations also rely on the processor arithmetic unit for address generation, the two use cases stimulate all DIFT-related hardware modules for various security policy configurations.

The next subsections describe these use cases and present the results obtained from the fault injection campaign. Tables 3 and 4 present detailed results for each use case. Each table highlights the DIFT-related registers, the fault types and the fault injection times leading to a successful attack. A tick indicates a successful attack for the corresponding register, fault type, and injection time. For registers larger than one bit, a grey line highlights a register bit sensitivity to the bit-flip fault type.

4.1 First use case: buffer overflow attack

The first use case is the exploitation of a buffer overflow leading to a potential Return-Oriented Programming (ROP) attack[1] and the execution of a shellcode. The attacker exploits a buffer overflow to reach the return address (*ra*) register. Due to the DIFT mechanism, the tag associated with the buffer data overwrites the *ra* register tag. Since the buffer data is manipulated by the user, it is tagged as *untrusted* (tag value = 1). When returning from the called function, the corrupted *ra* register is loaded into *PC* via a *jalr* instruction. The execution flow is hijacked and the first shellcode instruction is fetched from the address (*0x6fc*). This attack reveals the behaviour of DIFT when monitoring the PC tag.

Table 3 shows that 22 fault injections in 4 different DIFT-related registers can lead to a successful attack despite the DIFT mechanism (i.e., DIFT protection is bypassed). For example, it shows that a fault injection targeting the *pc_if_o_tag* register can defeat the DIFT protection if a fault is injected at cycle 3431 using a bit-flip or a set to 0 fault type. Furthermore, Table 3 shows that 5 different cycles can be targeted for the attack to succeed. In most cases, *bit-flip* leads to a successful injection with 11 successes over 22. Faults in *tpr_q* and *tcr_q* are successful since these registers maintain the propagation rules and the security policy configuration (see Table 1

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

Table 3: Buffer overflow: success per register, fault type and simulation time

	Cycle 3428			Cycle 3429			Cycle 3430			Cycle 3431			Cycle 3432		
	set0	set1	bitflip	set0	set1	bitflip	set0	set1	bitflip	set0	set1	bitflip	set0	set1	bitflip
pc_if_o_tag										✓		✓			
rf_reg[1]							✓		✓						
tcr_q	✓			✓			✓			✓			✓		
tcr_q[21]			✓			✓			✓			✓			✓
tpr_q	✓	✓		✓	✓										
tpr_q[12]			✓			✓									
tpr_q[15]			✓			✓									

and Table 2 for more details about each bit position). Both *pc_if_o_tag* and *rf_reg[1]* are also critical registers for this use case. Indeed, *pc_if_o_tag* allows the propagation of the PC tag while *rf_reg[1]* stores the tag of the return address register *ra*.

4.2 Second use case: format string attack

The second use case is a format string attack[2] overwriting the return address of a function to jump to a shellcode and starts its execution. This attack exploits the `printf()` function from the C library. It uses the `%u` and `%n` formats (see Chapter 12, Section 12.14.3 in [10] for detailed information) to write the targeted address. The format `%u` is used to print unsigned integer characters. The format `%n` is used to store in memory the number of characters printed by the `printf()` function, the argument it takes is a pointer to a signed int value. Let's call this value 'a'. 'a' is user-defined, so it is tagged as untrusted for DIFT computation. The vulnerable `printf` 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))`. The execution of such a `printf` leads to writing in memory 224 (0xe0) at address (a-4), 259 (0x103) at address (a-3), and 512 (0x200) at addresses (a-2) and (a-1). The attacker's objective is to overwrite the return address with 0x3e0 to call a shellcode. In this case, security policy prohibits the use of untrusted variables as store addresses. Since variable 'a' is untrusted, the DIFT protection raises an exception when storing a value at memory address `(a-4)`. This use case has been chosen to activate the load/store modes of the DIFT policy.

Table 4 shows that 52 fault injections in 10 DIFT-related registers can lead to a successful attack. Furthermore, it shows that 8 different cycles can be targeted for the attack to succeed. 29 successes over 52 are obtained with the *bit-flip* fault type. *alu_operand_a_ex_o_tag*, *alu_operand_b_ex_o_tag* and *alu_operator_o_mode* registers are critical during cycles 52477 and 52478 since they are used for tag propagation related to the C statement `(a-4)`. *alu_operand_a_ex_o_tag* and *alu_operand_b_ex_o_tag* sequentially store the tag associated to 'a' while *alu_operator_o_mode* stores the propagation rule according to the TPR configuration (see Table 1). *regfile_alu_waddr_ex_o_tag* stores the destination register index in which the tag resulting from tag propagation should be written. *check_s1_o_tag* maintains the TCR value from the decode stage to the execution stage, it is compared to the value of the operand tag for tag checking. *rf_reg[15]* stores the tag associated with the 'a' variable. *store_dest_addr_ex_o_tag* maintains the tag of the destination address during a store instruction in the execute stage. *use_store_ops_ex_o* drives a multiplexer to propagate the value stored in *store_dest_addr_ex_o_tag* register to the tag checking module. Finally, faults in *tpr_q* and *tcr_q* are successful since these registers maintain the propagation rules and the security policy configuration.

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

5 FAULTS PROPAGATION ANALYSIS

In this section, we present an in-depth analysis of the simulation results leading to successful attacks. The aim is to understand why an attack succeeds. For that purpose, we study the propagation of the fault through both temporal and logical views. Most of the faults targeting both TPR and TCR registers are not detailed in this section. Indeed, these faults mainly target the DIFT configuration and not the tag propagation and tag-checking computations. Faults targeting these registers can be performed in any cycle prior to their use.

5.1 First use case: buffer overflow attack

Figure 2 presents the *ra* register tag propagation in the context of the first use case for a non-faulty execution. It focuses on three clock cycles from the decoding of a `jalr` instruction (i.e., returning from the called function) to the DIFT exception due to a security policy violation. In cycle 3430, this tag is extracted from the *register file tag* (i.e., from *rf_reg[1]*). In cycle 3431, it is propagated to the *pc_if_o_tag* register. Then, in cycle 3432, it is propagated in the *pc_id_o_tag* register and the first shellcode instruction is decoded. Since *ra* is tagged as untrusted and the security policy prohibits the use of tagged data in PC (*Execute Check* bit = 1 in Table 2), an exception is raised during the tag check process, which is performed in parallel of the first shellcode instruction decoding.

Figure 2 illustrates the reason behind the sensitivity of registers *rf_reg[1]* and *pc_if_o_tag* at cycles 3430, 3431 and 3432 highlighted in Table 3. We can note that *pc_id_o_tag* register does not appear in Table 3 while Figure 2 shows its role during tag propagation. Actually, this register gets its value from *pc_if_o_tag*, so a fault injection in this register only delays the exception.

To further study the propagation of the fault, Figure 3 illustrates the logical relations between the DIFT-related registers (yellow boxes) and control signals or processor registers (grey boxes) driving the illegal instruction exception signal (red box). This figure does not describe the actual hardware architecture but highlights the logic path leading to an exception raise. An attacker performing fault injections would like to drive the exception signal to '0' to defeat the D-RI5CY DIFT solution. Figure 3 shows that a single fault could lead to a successful injection since all logic paths are built with *AND* gates. For instance, if register *rf_reg[1]* is set to 0, the tag

Table 4: Format string attack: success per register, fault type and simulation time

	Cycle 52477			Cycle 52478			Cycle 52479			Cycle 52480			Cycle 52481			Cycle 52482			Cycle 52483			Cycle 52484		
	set0	set1	bitflip	set0	set1	bitflip	set0	set1	bitflip	set0	set1	bitflip	set0	set1	bitflip	set0	set1	bitflip	set0	set1	bitflip	set0	set1	bitflip
alu_operand_a_ex_o_tag	✓		✓																					
alu_operand_b_ex_o_tag				✓		✓																		
alu_operator_o_mode	✓	✓		✓	✓																			
alu_operator_o_mode[0]			✓			✓																		
alu_operator_o_mode[1]			✓			✓																		
check_s1_o_tag																						✓		✓
regfile_alu_waddr_ex_o_tag[1]													✓											
rf_reg[15]																✓		✓	✓		✓			
store_dest_addr_ex_o_tag																						✓		✓
tcr_q	✓			✓			✓			✓			✓			✓			✓					
tcr_q[20]			✓			✓			✓			✓			✓			✓			✓			
tpr_q		✓			✓			✓			✓			✓										
tpr_q[12]			✓			✓			✓			✓			✓									
tpr_q[15]			✓			✓			✓			✓			✓									
use_store_ops_ex_o																						✓		✓

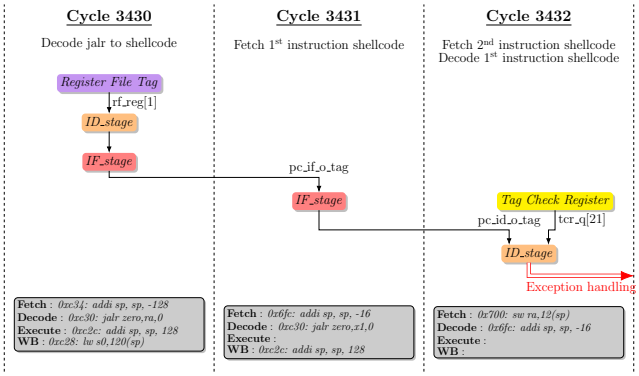

Figure 2: Tag propagation in a buffer overflow attack

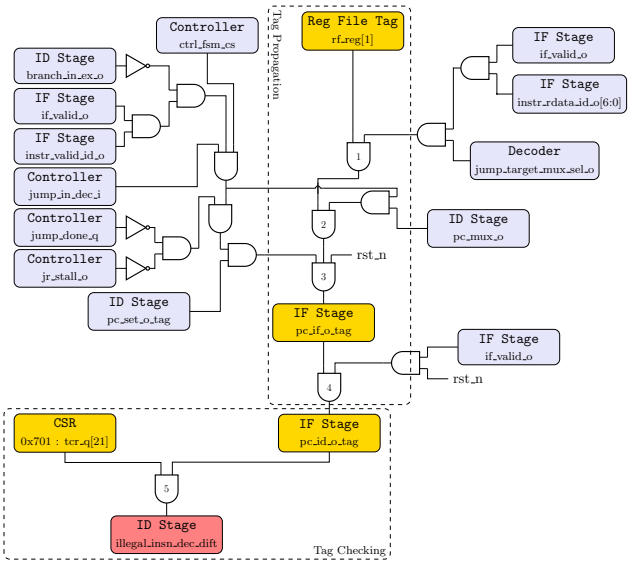

Figure 3: Logic description of the exception driving in a buffer overflow attack

will be propagated from *gate 1* to *gate 4*. Then, *gate 5* inputs are *tcr_q[21]* (i.e., '1') and *pc_id_o_tag* (i.e., '0', *gate 4* output). Thus, *gate 5* output is driven to '0', disabling the exception. From Figure 3, three fault propagation paths can be identified: from *gate 1* to *gate 5* if the fault is injected into *rf_reg[1]*, from *gate 4* to *gate 5* if a fault is injected into *pc_if_o_tag* and through *gate 5* if a fault is injected into either the *tcr_q* or *pc_id_o_tag*. Analysis of Figure 3 strengthens the results presented in Table 3 where *set to 0* and *bit-flip* fault types lead to successful attacks. The root cause is that the propagation paths consist entirely of *AND* gates.

5.2 Second use case: format string attack

Figure 4 details the tag propagation in the context of a format string attack case for a non-faulty execution and illustrates the reason behind the sensitivity of registers highlighted in Table 4. Figure 4 focuses on three clock cycles dedicated to the instruction `sw a4,0(a5)` decoding and execution which should lead to the storage of the value 224 at address (a-4). In cycles 52482 and 52483, `sw a4,0(a5)` is decoded and the source operands tag are retrieved from the tag register file. Particularly, the store destination address is retrieved from *rf_reg[15]* and stored in register *store_dest_addr_ex_o_tag*. In cycle 52484, the destination address of the store operation is computed by the processor Arithmetic Logic Unit (ALU). In parallel, *alu_operator_o_mode*, *alu_operand_a_ex_o_tag*, *alu_operand_b_ex_o_tag*, *store_dest_addr_ex_o_tag* and *check_s1_o_tag* registers drives the tag computation corresponding to the destination address. *use_store_ops_ex_o* drives a multiplexer to propagate the value stored in *alu_operand_a_ex_o_tag* register to the tag checking module. *alu_operand_a_ex_o_tag* and *alu_operand_b_ex_o_tag* sequentially store the tag associated to 'a' while *alu_operator_o_mode* stores the propagation rule according to the TPR configuration (see Table 1). *check_s1_o_tag* maintains the TCR value from the decode stage to the execution stage, it is compared to the value of the operand tag for tag checking. Then, the store should be executed in the Execute stage. However, the tag associated with the store destination address is set to 1 due to tag propagation (since it is computed from variable 'a'). Since the security policy prohibits the use of data tagged as *untrusted* as a store instruction destination address (*Load/Store Check* field of TCR = 1010), an exception is raised. *use_store_ops_ex_o*, highlighted in Table 4 but not shown in Figure 4, drives a multiplexer leading to the propagation of register *store_dest_addr_ex_o_tag*.

To further study the propagation of the fault, Figure 5 illustrates the logical relations between the DIFT-related registers (yellow

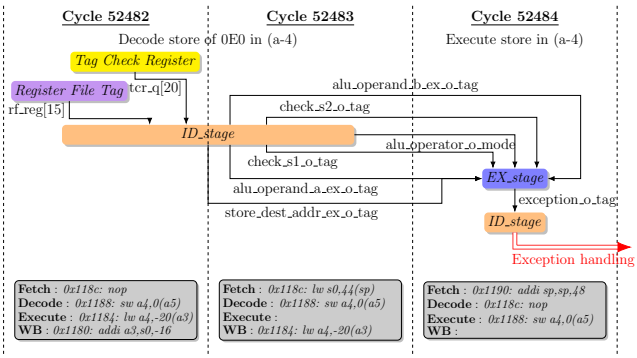

Figure 4: Tag propagation in a format string attack

Table 5: Logical fault injection simulation campaigns results

	Crash	NSTR	Delay	Success	Total
Buffer overflow	0	1380	20	22 (1.55%)	1422
WU-FTPd	0	1767	77	52 (2.74%)	1896

boxes) and control signals or processor registers (gray boxes) driving the illegal instruction exception signal (red box) for the second use case. Figure 5 shows that a single fault could lead to a successful injection since all logic paths are built with *AND* gates. For instance, if register *rf_reg[15]* is set to 0, this tag value will be propagated from *gate 8* to *gate 11* and to *mux 12*. Then, since *mux 12* output drives one *gate 3* input, *gate 3* output is driven to '0', the exception is disabled. From Figure 5, seven fault propagation paths can be identified: from *gate 1* to *gate 3* if the fault is injected into *tcr_q[20]*, through *gate 3* if a fault is injected into *check_s1_o_tag*, from *gate 4* or *gate 5* to *gate 3* if a fault is injected into *alu_operand_b_ex_o_tag* or *alu_operand_a_ex_o_tag*, from *mux 6* to *gate 3* if a fault is injected into *alu_operator_o_mode*, from *mux 7* to *gate 3* if a fault is injected into *regfile_alu_waddr_ex_o_tag*, from *gate 8* to *gate 3* if a fault is injected in the tag register file (i.e., register *rf_reg[15]*) and from *mux 11* to *gate 3* if a fault is injected in either *store_dest_addr_ex_o_tag* or *use_store_ops_ex_o*. Analysis of Figure 5 reinforces the results presented in Table 4 where *set to 0* and *bit-flip* fault types lead to successful attacks. As with the first use case, the main cause is that the propagation paths are fully made of *AND* gates. As shown in Table 4 *alu_operator_o_mode* register is sensitive to *set to 0* and *set to 1* fault types. Indeed, this register determines the tag propagation according to TPR. As stated in Section 3, the tag propagation is disabled when a TPR field is set to '00' and the output tag is set to 0 (i.e., trusted) when a TPR field is set to '11'.

6 DISCUSSION

In the previous sections, we extensively studied the behaviour of the D-RI5CY DIFT security mechanism for two use cases firstly in simulation and secondly through an architectural perspective from a temporal and logical side. Table 5 shows the overall results for each use case with regard to fault simulations' end status. The results show that we obtain 1.55% and 2.74% of successful fault injections.

A total of 3318 simulations have been performed. On average, about 2.23% of the fault injections lead to successful attacks. Among

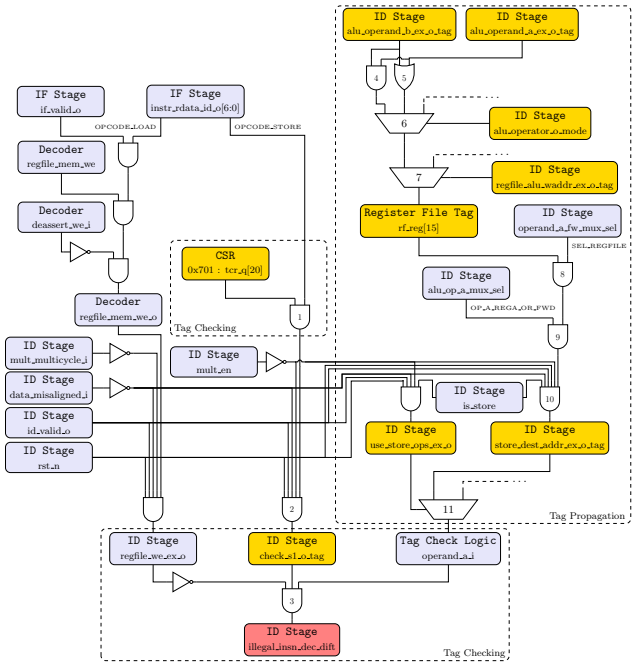

Figure 5: Logic description of the exception driving in a format string attack

the 74 successful injections, about 33.78% are due to set to 0 fault type, 12.16% are due to set to 1 fault type and 54.06% are due to a single bit-flip. Despite these cases do not lead to a DIFT bypass, it is also interesting to highlight that 2.92% of the simulated injections delay the DIFT exception. This analysis demonstrates that the DIFT is vulnerable to FIA and propagation of faults is facilitated by combinatorial paths fully made of AND gates.

Finally, the proposed analysis highlights that, including the entire tag register file, 42 DIFT-related registers can be targeted to bypass the D-RI5CY DIFT protection. This detailed information is precious to build efficient and lightweight countermeasures in a low-power processor for IoT devices.

7 CONCLUSION

This work studies the vulnerability of the D-RI5CY DIFT mechanism to fault injection attacks. A cycle-accurate bit-accurate simulated fault injections campaign has been performed to identify sensitive DIFT-related registers and determine specific time locations an attacker could target. Moreover, we proposed an in-depth analysis of the simulation results leading to successful attacks. The proposed analysis demonstrates that the propagation of faults is facilitated by paths fully made of AND gates.

In future work, we plan to implement countermeasures into the D-RI5CY DIFT mechanism to counter fault injection attacks and evaluate these countermeasures taking into account constraints such as performance, area overhead and security. We plan to focus on simple parity and Hamming code. We will compare these different lightweight protections, able to detect and/or correct faults in terms of security, area overhead and performance. We also plan to take into account a more complex threat model such as the multi-fault model.

REFERENCES

[1] Christopher Brant, Prakash Shrestha, Benjamin Mixon-Baca, Kejun Chen, Said Varlioglu, Nelly Elsayed, Yier Jin, Jedidiah Crandall, and Daniela Oliveira. 2021. Challenges and Opportunities for Practical and Effective Dynamic Information Flow Tracking. *ACM Computing Surveys (CSUR)*. DOI: 10.1145/3483790.

[2] Kejun Chen, Xiaolong Guo, Qingxu Deng, and Yier Jin. 2021. Dynamic Information Flow Tracking: Taxonomy, Challenges, and Opportunities. *Micromachines*. DOI: 10.3390/mi12080898.

[3] Kejun Chen, Lei Sun, and Qingxu Deng. 2022. Hardware and Software Co-verification from Security Perspective in SoC Platforms. *Journal of Systems Architecture*. DOI: 10.1016/j.sysarc.2021.102355.

[4] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Association for Computing Machinery. DOI: 10.1145/1250662.1250722.

[5] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware Information Flow Tracking. *ACM Computing Surveys*. DOI: 10.1145/3447867.

[6] Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2009. Decoupling Dynamic Information Flow Tracking with a Dedicated Coprocessor. In *International Conference on Dependable Systems & Networks*. IEEE. DOI: 10.1109/DSN.2009.5270347.

[7] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. 2013. Hardware Designer's Guide to Fault Attacks. *IEEE Transactions on Very Large Scale Integration Systems*. DOI: 10.1109/TVLSI.2012.2231707.

[8] Vanthanh Khuat, Jean-Max Dutertre, and Jean-Luc Danger. 2021. Analysis of a Laser-induced Instructions Replay Fault Model in a 32-bit Microcontroller. In *Digital System Design (DSD)*. DOI: 10.1109/DSD53832.2021.00061.

[9] Johan Laurent, Vincent Beroulle, Christophe Deleuze, and Florian Pebay-Peyroula. 2019. Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures. In *Design, Automation & Test in Europe Conference (DATE)*. DOI: 10.23919/DATE.2019.8715158.

[10] Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. 2023. The GNU C Library Reference Manual. https://www.gnu.org/s/libc/manual/pdf/libc.pdf.

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

[12] Christian Palmiero, Giuseppe Di Guglielmo, Luciano Lavagno, and Luca P. Carloni. 2018. Design and Implementation of a Dynamic Information Flow Tracking Architecture to Secure a RISC-V Core for IoT Applications. In *High Performance Extreme Computing*. DOI: 10.1109/HPEC.2018.8547578.

[13] Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. 2021. Fiver – robust verification of countermeasures against fault injections. *IACR Transactions on Cryptographic Hardware and Embedded Systems*. DOI: 10.46586/tches.v2021.i4.447-473.

[14] Marvin Saß, Richard Mitev, and Ahmad-Reza Sadeghi. 2023. Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M. In *USENIX security symposium*.

[15] Robert Schilling, Pascal Nasahl, Martin Unterguggenberger, and Stefan Mangard. 2022. SFP: Providing System Call Flow Protection against Software and Fault Attacks. In *Workshop on Hardware and Architectural Support for Security and Privacy* (HASP '22). Association for Computing Machinery.

[16] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. *SIGPLAN Not.* DOI: 10.1145/1037187.1024404.

[17] Niek Timmers, Albert Spruyt, and Marc Witteman. 2016. Controlling PC on ARM Using Fault Injection. In *Fault Diagnosis and Tolerance in Cryptography (FDTC)*. DOI: 10.1109/FDTC.2016.18.

[18] Thomas Trouchkine, Sébanjila Kevin K Bukasa, Mathieu Escouteloup, Ronan Lashermes, and Guillaume Bouffard. 2021. Electromagnetic Fault Injection Against a Complex CPU, toward new Micro-architectural Fault Models. *Journal of Cryptographic Engineering*. DOI: 10.1007/s13389-021-00259-6.

[19] Loïc Zussa, Jean-Max Dutertre, Jessy Clédière, Bruno Robisson, and Assia Tria. 2012. Investigation of timing constraints violation as a fault injection means. In *27th Conference on Design of Circuits and Integrated Systems (DCIS)*. Avignon, France, (Nov. 2012), pas encore paru. https://hal-emse.ccsd.cnrs.fr/emse-00742652.