COLLEGE MATHSTIC
DOCTORAL BRETAGNE
BRETAGNE OCEANE

Université Bretagne Sud

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE BRETAGNE SUD

ÉCOLE DOCTORALE N° 644
Mathématiques et Sciences et Technologies de l'Information et de la Communication en Bretagne Océane
Spécialité : *Informatique et Architectures Numériques*

Par
William PENSEC

Protection d'un processeur avec DIFT contre des attaques physiques

« Sous-titre de la thèse »

Thèse présentée et soutenue à Lorient, le //2024
Unité de recherche : Université Bretagne Sud, UMR CNRS 6285, Lab-STICC
Thèse N° : « si pertinent »

Rapporteurs avant soutenance :

Prénom NOM Fonction et établissement d'exercice
Prénom NOM Fonction et établissement d'exercice
Prénom NOM Fonction et établissement d'exercice

Composition du Jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition du jury doit être revue pour s'assurer qu'elle est conforme et devra être répercutée sur la couverture de thèse

Président :	Prénom NOM	Fonction et établissement d'exercice *(à préciser après la soutenance)*
Examinateurs :	Prénom NOM	Fonction et établissement d'exercice
	Prénom NOM	Fonction et établissement d'exercice
	Prénom NOM	Fonction et établissement d'exercice
	Prénom NOM	Fonction et établissement d'exercice
Dir. de thèse :	Guy GOGNIAT	Professeur des Universités (Lab-STICC, Université Bretagne Sud)
Co-dir. de thèse :	Vianney LAPÔTRE	Maitre de Conférence HDR (Lab-STICC, Université Bretagne Sud)

Invité(s) :

Prénom NOM Fonction et établissement d'exercice

$\mathcal{A}$d mentes inquisitivas quae lucem futuri Scientiae accendunt.

Aux esprits curieux qui illuminent l'avenir de la Connaissance.

To the inquisitive minds that are lighting up the future of Knowledge.

Remerciements

Je tiens à remercier
I would like to thank. my parents..
J'adresse également toute ma reconnaissance à ....
....

Table of Contents

Acronyms

CSR	Control and Status Registers
DIFT	Dynamic Information Flow Tracking
FIA	Fault Injection Attack
RA	Return Address
TCR	Tag Check Register
TPR	Tag Propagation Register

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

IoT without security means Internet of Threats

Stéphane Nappo

Contents

1.1 Context

1.2 Motivations

1.3 Objectives

1.4 Manuscript outline

This work is segmented in seven chapters, the first being this introduction.

Chapter 2

Chapter 3 presents the background of this work with the presentation of the RISC-V ISA, the architecture of the D-RI5CY core and the DIFT works. Then, the use cases used in this work are going to be presented. Finally, a vulnerability assessment will be done to show how these use case are vulnerable against FIA and where.

Chapter 4 introduces a new tool to automatise fault injection campaigns in simulation. This tool, FISSA, allows a designer to assess his design during the conception phase. This chapter will present how it works and how to use it and compares it to others tool available in the litterature.

Chapter 5 details the different implementation of countermeasures to protect the D-RI5CY core against FIA and evaluate these protections in terms of area, performance and efficiency.

Chapter 6

Chapter 7

Chapter 2

State of the Art

Contents

2.1 Introduction

This chapter provides an overview of related work to contextualize the primary objectives of this thesis. Firstly, Information Flow Tracking (IFT) is introduced, detailing the different types and their respective purposes. I will discuss the various levels of monitoring, from program behaviour to the detection of hardware trojans. Subsequently, Physical Attacks are examined, focusing on two main types: Side-Channel Attacks (SCA) and Fault Injection Attacks (FIA). Finally, as my research will concentrate on FIA, I will exclusively present countermeasures against Fault Injection Attacks.

2.2 Information Flow Tracking

This section presents the various types of IFT and the different functional levels associated with Dynamic IFT.

2.2.1 Different types of IFT

There are two distinct types of IFT approaches: static and dynamic, each with its own specific objectives.

2.2.1.1 Static IFT (SIFT)

This approach involves analysing the flow of information within a system without actually executing the program. The goal of static IFT is to determine potential information flows and data pathways by

examining the codebase or system architecture. This method is particularly useful for identifying theoretical vulnerabilities and ensuring compliance with design principles before deployment. Static analysis is comprehensive as it covers all possible execution paths, but it may also generate false positives by flagging theoretical flows that might not occur in practice.

2.2.1.2 Dynamic IFT (DIFT)

In contrast, dynamic IFT tracks information flow in real-time as the system operates. This method observes how data actually moves through the system under various operating conditions, providing a practical and immediate understanding of information handling and leakage. The goal of dynamic IFT is to detect and respond to security breaches or compliance issues as they happen, offering a real-world perspective on the system's security posture. However, this approach might not cover all potential data paths as it is dependent on the specific conditions and inputs provided during the monitoring period.

2.2.2 Different levels of IFT

2.2.2.1 Application level

2.2.2.2 OS level

2.2.2.3 Architecture level

2.2.2.4 Gate level

2.2.3 DIFT Architectures

2.2.3.1 Off-Core

2.2.3.2 Off-Loading

2.2.3.3 In-Core

2.3 Physical Attacks

2.3.1 Side-Channel Attacks

2.3.2 Fault Injection Attacks

2.4 Countermeasures against FIA

Chapter 3

Dynamic Information Flow Tracking - Vulnerabilities Assessment

Contents

This chapter provides the background of this thesis and the vulnerability assessment. The first section offers a description of the RISC-V Instruction Set Architecture (ISA) and an overview of the specific RISC-V design under consideration. The second section details and describe the considered uses cases of this thesis. The third section will then detail a vulnerabilities assessment of the D-RI5CY, using these two case studies.

3.1 D-RI5CY

In this section, we describe the RISC-V ISA and detail the DIFT design we have chosen to focus on.

RISC-V Instruction Set Architecture (ISA) is an open and free ISA, which was originally developped at University of California, Berkeley, in 2010, and now is managed and supported by the RISC-V Foundation, having more than 70 members including companies such as Google, AMD, Intel, etc. The architecture was designed with a focus on simplicity and efficiency, embodying the Reduced Instruction Set Computer (RISC) principles. Unlike proprietary ISAs, RISC-V is freely available for anyone to

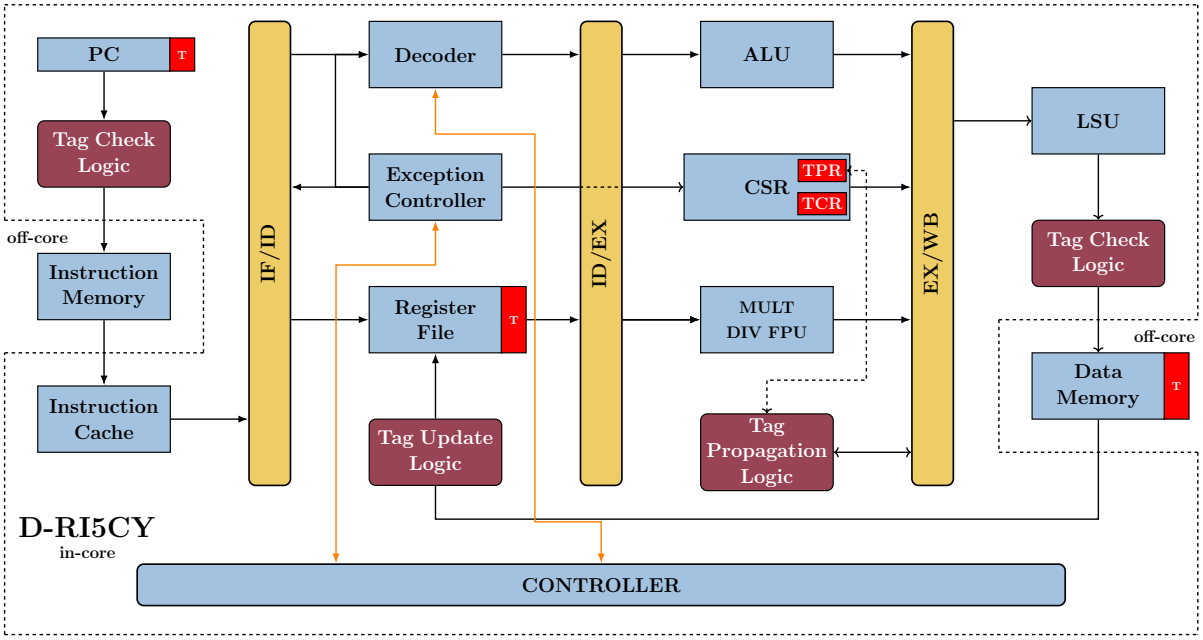


Figure 3.1: D-RI5CY processor architecture overview. DIFT-related modules are highlighted in red.

use without licensing fees, making it a popular choice for academic research, commercial products, and educational purposes.

Technically, RISC-V features a modular design, allowing developers to incorporate only the necessary components for their specific application, which can significantly reduce the processor's complexity and power consumption. It supports several base integer sets classified by width—mainly RV32I, RV64I, and RV128I for 32-bit, 64-bit, and 128-bit architectures respectively. Each base set can be extended with additional modules for applications requiring floating-point computations (e.g., RV32F, RV64F), atomic operations (e.g., RV32A, RV64A), and more. This modularity and the openness of RISC-V have spurred a wide range of innovations in processor design and applications in areas ranging from embedded systems to high-performance computing.

DIFT design We have selected the D-RI5CY design, which utilises the RI5CY core supported by PULPino and developed by ETH Zurich. This is a 4-stage, in-order, 32-bit RISC-V core optimised for low-power embedded systems and IoT applications. It fully supports the base integer instruction set (RV32I), compressed instructions (RV32C), and the multiplication instruction set extension (RV32M) of the RISC-V ISA. Additionally, it includes a set of custom extensions (RV32XPulp) that support hardware loops, post-incrementing load and store instructions, and, ALU and MAC operations.

D-RI5CY [1] has been developed by researchers of Columbia University, in the USA, in partnership with Politecnico di Torino, in Italy. D-RI5CY use the RI5CY processor in which they implemented a hardware in-core DIFT.

Figure 3.1 presents an overview of the D-RI5CY processor's architecture. In red are represented the DIFT specific modules. These modules allow storing, propagating and checking tags during the execution

Table 3.1: Instructions per security classes

Class	**Instructions**
Load/Store	*LW, LH[U], LB[U], SW, SH, SB, LUI, AUIPC, XPulp Load/Store*
Logical	*AND, ANDI, OR, ORI, XOR, XORI*
Comparison	*SLTI, SLT*
Shift	*SLL, SLLI, SRL, SRLI, SRA, SRAI*
Jump	*JAL, JALR*
Branch	*BEQ, BNE, BLT[U], BGE[U]*
Integer Arithmetic	*ADD, ADDI, SUB, MUL, MULH[U], MULHSU, DIV[U], REM[U]*

of a sensitive application. 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.

Authors defined a library of routines to initialise the tags of the data coming from potentially malicious channels. Additionally, they extended the RI5CY ISA with memory and register tagging instructions. The default 1-bit tag is "*0*", this means that the data is trusted, otherwise, the tag would be set to "*1*" which means that the data is untrusted. Moreover, they augmented the program counter with a tag of one bit and the register file with one tag per register's byte (marked as T). Finally, they added a tag in the data memory, however, it is a 4-bits tag. Each data element is physically stored in memory with its associated tag.

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.

In the Control and Status Registers (CSR), they added two additionals 32-bits registers : Tag Propagation Register (TPR) and Tag Check Register (TCR). These registers are used to store the security policy for both propagation and check. These registers contains a default policy but they can be modified during runtime with a simple instruction. These policies consist of rules, which have fine-grain control over tag propagation and check for different classes of instructions. The rules specify how the tags of the instruction operands are combined and checked. Table 3.1 shows the different instructions for each category represented in both TPR and TCR.

Table 3.2 shows the TPR configurations for the security policies considered in our work. 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 enabling the

Table 3.2: 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 1	0 0 1	1 0	1 0	0 0	1 0	1 0	0 0	1 0
Policy 2	1 1 1	1 0	1 0	1 0	1 0	1 0	1 0	1 0

Table 3.3: 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
Policy V2	0	0 0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0	0 1 1

source, source-address, and destination-address tags, respectively.

Table 3.3 shows the TCR configurations considered in our work. Each instruction type has a user-configurable 3-bits tag control policy field (except for *Execute Check*, *Branch Check* and *Load/Store Check* which have 1, 2 and 4-bits 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-bits 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.

Figure 3.2 shows a representation of the DIFT work flow. At first ①, the D-RI5CY initalises the configuration registers (TPR and TCR) from the default security policy and tag all others registers . Then at program startup ②, D-RI5CY initialises all the tags to *trusted* (i.e, set at 0). The tag propagation ③ and verification ④ happen in the D-RI5CY pipeline in parallel with the standard behavior, without incurring any latency overhead.

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 user manipulates the buffer. 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).

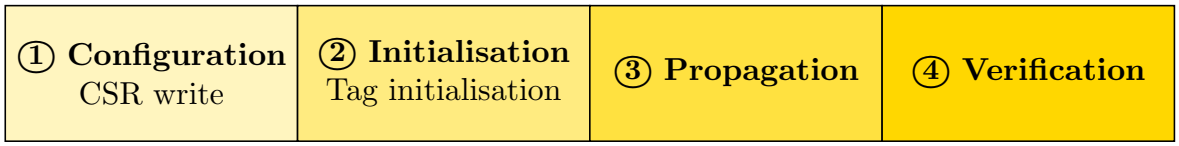


Figure 3.2: Representation of the DIFT flow.

3.2 Use cases

This section details the considered use cases in our work. The first two use cases come from the original paper [1]. The third use case is a home-made case which is used to analyse the different DIFT part not studied in others use cases.

3.2.1 First use case: Buffer Overflow

The first use case involves exploiting a buffer overflow, potentially leading to a Return-Oriented Programming (ROP) attack[1] and the execution of a shellcode. The attacker exploits the buffer overflow to access the return address (*RA*) register. When the function returns, the corrupted *RA* register is loaded into the *PC* via a *jalr* instruction. This hijacks the execution flow, causing the first shellcode instruction to be fetched from address (*0x6fc*). Due to the DIFT mechanism, the tag associated with the buffer data overwrites the *RA* register tag. As the buffer data is user-manipulated, it is tagged as *untrusted* (tag value = 1). Consequently, when the first shellcode instruction is fetched, the tag associated with the *PC* propagates through the pipeline until the DIFT mechanism detects a violation of the security policy and raises an exception. This attack demonstrates the behaviour of DIFT when monitoring the *PC* tag. This use case employs the first security policy from Table 3.2 and Table 3.3.

Listing 3.1 displays the C code for the buffer overflow scenario. The assembly code on line 22 of this listing represents the saving of the register *x8*, which is the *saved register 0* or *frame pointer* register in the RISC-V ISA. Next, the source buffer is filled with A's characters and the shellcode address is appended to the end of this source buffer. Finally, lines 30-33 illustrate the tag initialisation on the source buffer.

Figure 3.3 represents the five steps from the source buffer initialisation to the first shellcode instruction being fetched. In Figure 3.3a, the source buffer is initialised with A's, the destination buffer is empty, and both *PC* and *RA* register are trusted. In the second step, Figure 3.3b, the source buffer is copied into the destination buffer, and in Figure 3.3c, the *RA* register is compromised with the address of the shellcode function. At the same steps, during the buffer copy, the associated tags are copied into the destination buffer. Finally, in Figure 3.3d, the *PC* loads the *RA* register along with its tag. The *PC* loses its integrity. In Figure 3.3e, the *PC* address is fetched, and the instruction is sent into the pipeline along with the tag.

3.2.2 Second use case: Format String (WU-FTPd)

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 use case use the first security policy from Table 3.2 and Table 3.3.

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`

Listing 3.1: Buffer overflow C code

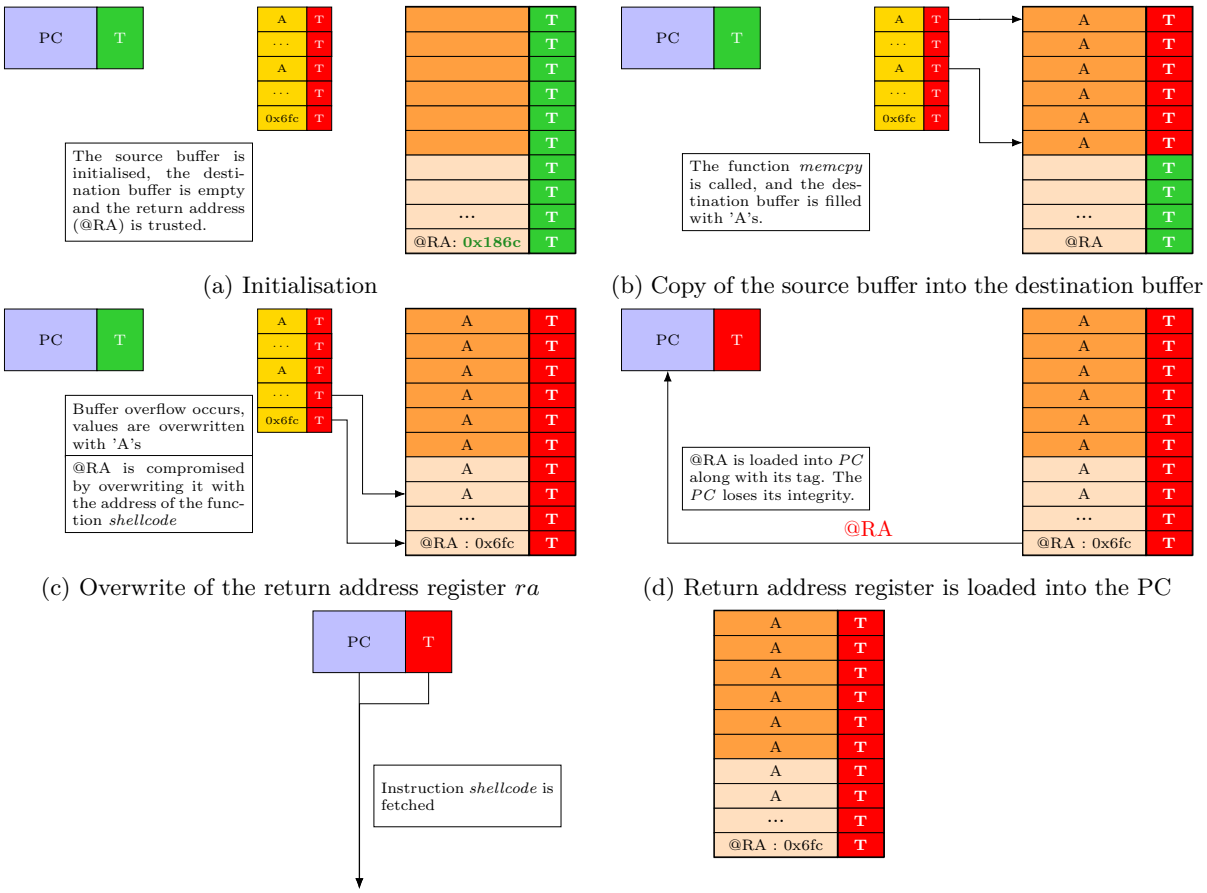
```
1  #define BUFSIZE 16
2  #define OVERFLOWSIZE 256
3
4  int base_pointer_offset;
5  long overflow_buffer[OVERFLOWSIZE];
6
7  int shellcode() {
8      printf("Success !!\n");
9      exit(0);
10 }
11
12 void vuln_stack_return_addr(){
13     long *stack_pointer;
14     long stack_buffer[BUFSIZE];
15     char propolice_dummy[10];
16     int overflow;
17
18     /* Just a dummy pointer setup */
19     stack_pointer = &stack_buffer[1];
20
21     /* Store in i the address of the stack frame section dedicated to function arguments */
22     register int i asm("x8");
23
24     /* First set up overflow_buffer with 'A's and a new return address */
25     overflow = (int)((long)i - (long)&stack_buffer);
26     memset(overflow_buffer, 'A', overflow-4);
27     overflow_buffer[overflow/4-1] = (long)&shellcode;
28
29     /* TAG INITIALISATION */
30     for(int j=0; j<overflow/4; j++) {
31         asm volatile ("p.spsw x0, 0(%[ovf]);"
32                       ::[ovf] "r" (overflow_buffer+j));
33     }
34
35     /* Then overflow stack_buffer with overflow_buffer */
36     memcpy(stack_buffer, overflow_buffer, overflow);
37
38     return;
39 }
40
41 int main(){
42     vuln_stack_return_addr();
43     printf("Attack prevented.\n");
44     return EXIT_SUCCESS;
45 }
```

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 [2] for detailed information) to write the targeted address.

Listing 3.2 shows the C code of this use case. The `echo` function assign the *x8* register to a variable '`i`' which goes into another variable '`a`'. The lines 13-14 are used to initialise the tag associated to the variable '`a`'. This variable '`a`' is user-defined, so it is tagged as untrusted for DIFT computation. The vulnerable statement is the `printf` statement in line 16. 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.

The execution of the `printf` at line 16 leads to write in memory 224 (0xe0) at address (a-4), 224+35 so 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*' which represent the address of the first function, called *secretFunction* in Listing 3.2. 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 3.4 represents the different steps to overwrite the memory with the exact address of the malicious function. We can see that after each write and the right shift of the writing, the address appears. Finally, we have the address '*000002000003E0*' in memory from 'A+2' to 'A-4' but as an address is on 32-bits in our architecture, the address fetched by the pipeline is only '*000003E0*'.



(a) Initialisation

(b) Copy of the source buffer into the destination buffer

(c) Overwrite of the return address register *ra*

(d) Return address register is loaded into the PC

(e) PC address instruction is fetched

Figure 3.3: Representation of how the ROP attack works

3.2.3 Third use case: Compare/Compute

The third use case implements security policy 2 from Table 3.2 and Table 3.3. This use case has been developed for our use but does not induce a real attack as both previous use cases. Policy 2 allows tag propagation for each type of instruction, unlike 1 which disables propagation for branch and comparison instructions and for *Load/Store Enable* field. Regarding TCR configuration, 2 only enables tag checks for arithmetic instructions from input sources. Listing 3.3 shows the C code used for this use case. Lines 2 to 4 initialize variables, lines 5 and 6 configure the new security policy by writing in the TPR and TCR registers with an assembly line and the write in the CSR instruction. Line 7 tags the variable `a` as untrusted (tag is set to "*0*"). In line 8, variables `a` and `b` are compared to determine which arithmetic operation should be performed. Lines 9 to 21 detail the assembly code generated from the line 8 C statement. It executes the operations according to the values of `a` and `b` stored in the registers `a4` and `a5` respectively. The `(a>b)` condition and its associated branch is computed in line 9, the `(a-b)` subtraction in line 14 and the `(a+b)` addition in line 20. Policy 2 focuses on arithmetic operations. It prohibits the use of untrusted variables for arithmetic computations. Since variable `a` is untrusted and `(b>a)`, the DIFT

Listing 3.2: WU-FTPd C code

```
1  void secretFunction(){
2      printf("Congratulations!\n");
3      printf("You have entered in the secret function!\n");
4
5      exit(0);
6  }
7
8  void echo(){
9      int a;
10     register int i asm("x8");
11     a = i;
12
13     asm volatile ("p.spsw x0, 0(%[a]);"
14                   ::[a] "r" (&a));
15
16     printf("%224u%n%35u%n%253u%n%n", 1, (int*) (a-4), 1, (int*) (a-3), 1, (int*) (a-2), (int*) (a-1));
17
18     return;
19 }
20
21 int main(int argc, char* argv[]){
22     volatile int a = 1;
23
24     if(a)
25         echo();
26     else
27         secretFunction();
28
29     return 0;
30 }
```

Table 3.4: Memory overwrite

Adresse	A-4	A-3	A-2	A-1	A	A+1	A+2
A-4	*0xE0*	*0x00*	*0x00*	*0x00*			
A-3		*0x03*	*0x01*	*0x00*	*0x00*		
A-2			*0x00*	*0x02*	*0x00*	*0x00*	
A-1				*0x00*	*0x02*	*0x00*	*0x00*
Memory	0xE0	0x03	0x00	0x00	0x02	0x00	0x00

protection raises an exception when executing instruction `add a5,a4,a5` (i.e., the `a+b` C statement).

3.3 Vulnerability assessment

In order to analyse the behaviour of the processor at application runtime against Fault Injection Attacks, we have simulated some fault injections campaigns in which we inject fault inside the 55 registers associated to the DIFT, which correspond to 127 bits in total. Table 3.5 shows the repartition of these registers in every pipeline stage of the RI5CY core and the number of associated bits. This work has been published in a conference [3].

We assess the design with fault injection campaigns. With their results associated, we can deduce which registers are vulnerables with the cycle associated and the fault model. This assessment is done for each use case for a more precise analysis and to understand how the tag is propagated and checked before the exception.

3.3.1 Fault model for vulnerability assessment

In this vulnerability assessment, we consider an attacker able to inject faults into DIFT-related regiters leading to *set to 0*, *set to 1*, and *single bit-flip in one register at a given clock cycle*. To bypass the DIFT

Listing 3.3: Compare/Compute C Code

```
int main(){
   int a, b = 5, c;
   register int reg asm("x9");
   a = reg;
   asm volatile("csrw 0x700, tprValue");
   asm volatile("csrw 0x701, tcrValue");
   asm volatile("p.spsw x0, 0(\%0);" :: "r" (&a));
   c = (a > b) ? (a-b) : (a+b);
       //42c:   ble a4,a5,448
       //430:   addi a5,s0,-16
       //434:   lw a4,-12(a5)
       //438:   addi a3,s0,-16
       //43c:   lw a5,-4(a3)
       //440:   sub a5,a4,a5
       //444:   j 45c
       //448:   addi a5,s0,-16
       //44c:   lw a4,-12(a5)
       //450:   addi a3,s0,-16
       //454:   lw a5,-4(a3)
       //458:   add a5,a4,a5
       //45c:   sw a5,-24(s0)
   return EXIT_SUCCESS;
}
```

Table 3.5: Numbers of registers and quantity of bits represented

HDL Module	Number of registers	Number of bits in registers
Instruction Fetch Stage	2	2
Instruction Decode Stage	14	19
Register File Tag	1	32
Execution Stage	1	1
Control and Status Registers	2	64
Load/Store Unit	4	9
Total	**24**	**127**

mechanism, the main attacker's goal is to prevent an exception being raised. To reach this objective, any DIFT-related register maintening tag value, driving the tag propagation or the tag update process or maintening the security policy configuration can be targeted.

3.3.2 First use case: Buffer overflow

Table 3.6 shows that 22 fault injections in four 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 3.6 shows that five 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 3.2 and Table 3.3 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*.

Now that we have these results, we can analyse them and 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

Table 3.6: 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]			✓			✓									

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.

Figure 3.4 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 3.3), an exception is raised during the tag check process, which is performed in parallel of the first shellcode instruction decoding.

Figure 3.4 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 3.6. We can note that *pc_id_o_tag* register does not appear in Table 3.6 while Figure 3.4 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 3.5 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 3.5 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[1]* is set to 0, the tag 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 3.5, 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 3.5 strengthens the results presented in Table 3.6 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.

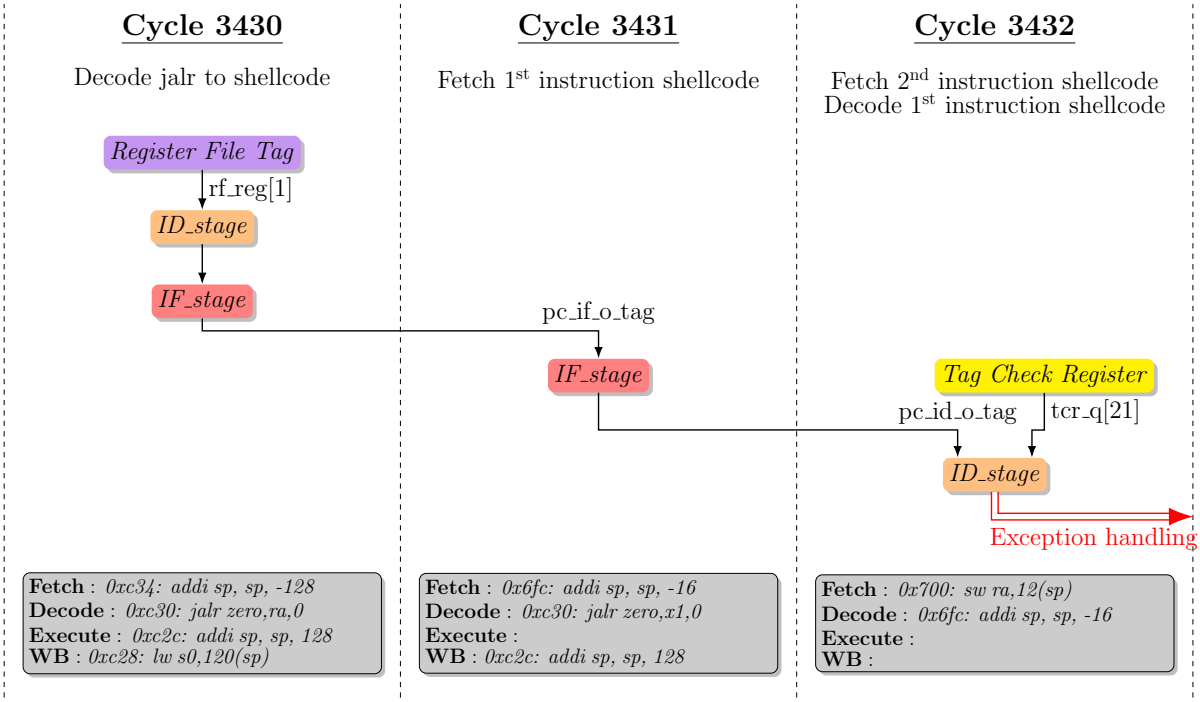


Figure 3.4: Tag propagation in a buffer overflow attack

3.3.3 Second use case: Format string (WU-FTPd)

Table 3.7 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 3.2). *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. The last two registers, *tpr_q* and *tcr_q* are critical when we fault the bit 12 of TPR because the load/store mode which is set to *10* but if we change it the propagation policy will change and then the tag will not be propagated as a mode set to *11* will clear the tag. A bit-flip at bit 15 will impact the behaviour as it stores the load/store enable source tag. Finally, bit 20 of TCR store the load/store check destination address tag which is used when the program wants to store at the address (a-4).

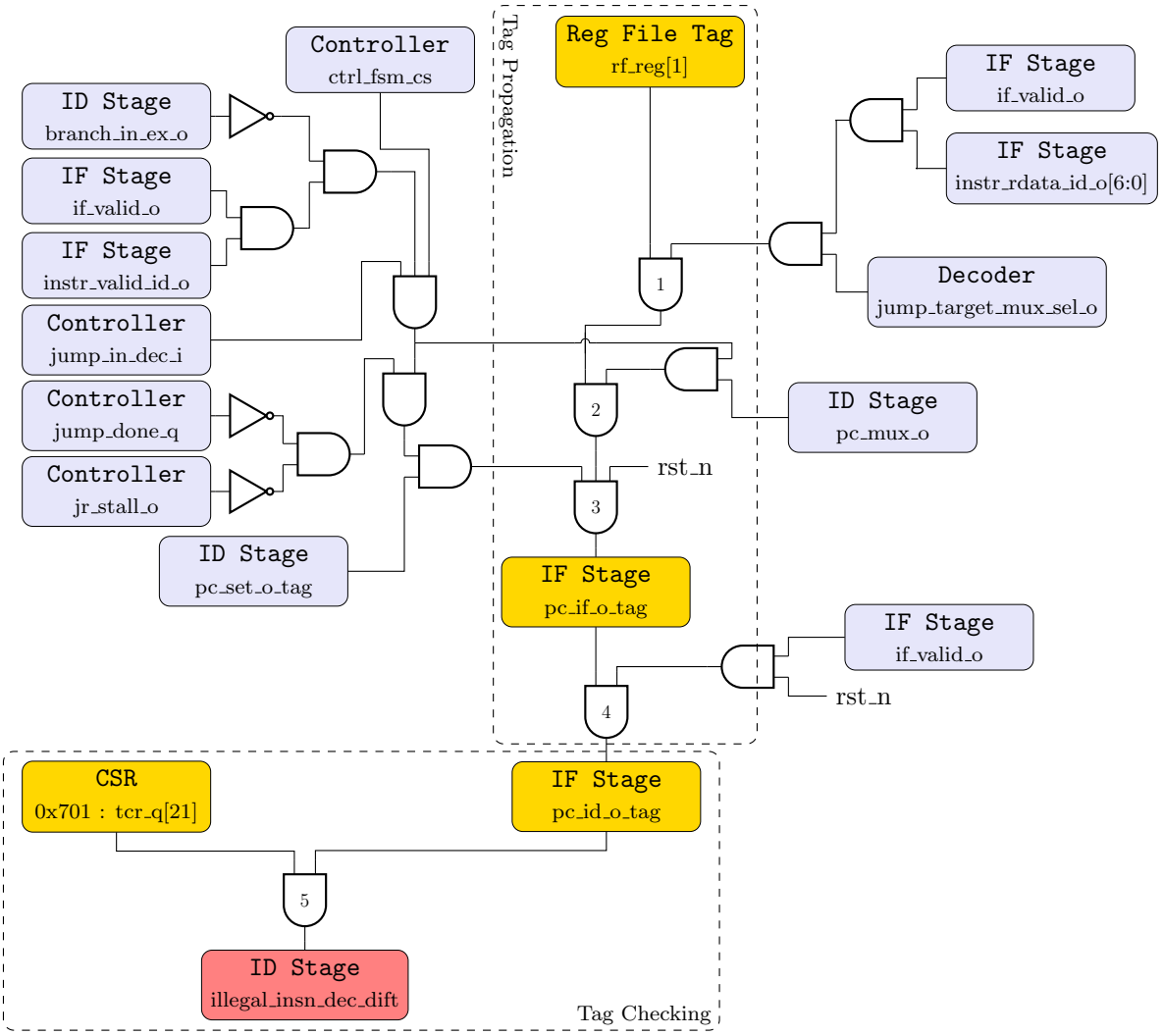


Figure 3.5: Logic description of the exception driving in a buffer overflow attack

Table 3.7: 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 3.6 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 3.7. Figure 3.6 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 3.2). *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 3.7 but not shown in Figure 3.6, drives a multiplexer leading to the propagation of register *store_dest_addr_ex_o_tag*.

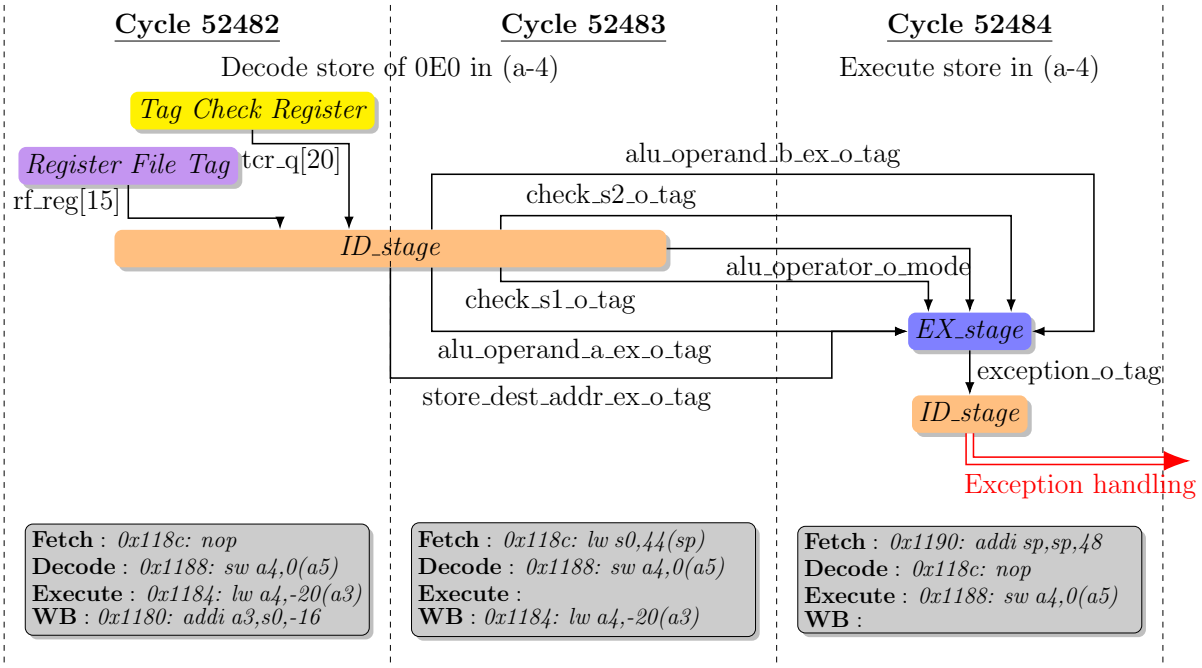


Figure 3.6: Tag propagation in a format string attack

To further study the propagation of the fault, Figure 3.7 illustrates the logical relations between the

DIFT-related registers (yellow boxes) and control signals or processor registers (gray boxes) driving the illegal instruction exception signal (red box) for the second use case. Figure 3.7 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 3.7, 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 3.7 reinforces the results presented in Table 3.7 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 3.7 *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. 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'.

3.3.4 Third use case: Compare/Compute

Table 3.8 shows that 19 fault injections in 6 DIFT-related registers can lead to a successful attack. Furthermore, it shows that 4 different cycles can be targeted for the attack to succeed. The highest success rate is obtained with the *bit-flip* fault type with 10 successes over 19. Faults in *rf_reg[14]* and *alu_operand_a_ex_o_tag* are successful since these registers store the tag associated to variable `a` during tag propagation. *check_s1_o_tag* maintains one configuration bit from *tcr_q* during tag checking. *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. For this case, the critical registers can be found in previous case, *alu_operand_a_ex_o_tag* propagate the tag of the taggued variable in the code (variable `a`). Finally, observations for both *tpr_q* and *tcr_q* are similar that for previous case studies. Finally, faults in *tpr_q* and *tcr_q* are successful since these registers maintain the propagation rules and the security policy configuration.

Figure 3.8 focuses on the three cycles, represented in red, corresponding to `add a5,a4,a5` instruction (C statement `(a+b)`) decoding and execution in the context of the third use case. The instruction `add a5,a4,a5` is in decode stage during cycles 833 and 834 and the tag associated to the untrusted variable `a` is retrieved from *rf_reg[14]*. In cycle 835, this addition is executed. In parallel, variable `a` tag is propagated to the tag check logic unit, which behavior is driven by *check_s1_o_tag* throught *alu_operand_a_ex_o_tag*. Since the V2 security policy prohibits the use of untrusted data as a source operand of an arithmetic operation, an exception is raised.

Figure 3.8 illustrates the reason behind the sensitivity of registers *rf_reg[14]*, *alu_operand_a_ex_o_tag* and *check_s1_o_tag* highlighted in Table 3.8. Note that *use_store_ops_ex_o* does not appear in Figure 3.8. This register drives a multiplexer leading to tag propagation presented in Figure 3.8.

To further study the faults propagation, Figure 3.9 illustrates the logical relations between the DIFT-related registers (yellow boxes) and control signals or processor registers (gray boxes) driving the illegal

Table 3.8: Compare/compute: number of faults per register, per fault type and per cycle

	Cycle 832			Cycle 833			Cycle 834			Cycle 835		
	set0	set1	bitflip	set0	set1	bitflip	set0	set1	bitflip	set0	set1	bitflip
alu_operand_a_ex_o_tag										✓		✓
check_s1_o_tag										✓		✓
rf_reg[14]				✓		✓	✓		✓			
tcr_q	✓			✓			✓					
tcr_q[0]			✓			✓			✓			
tpr_q		✓										
tpr_q[12]			✓									
tpr_q[15]			✓									
use_store_ops_ex_o											✓	✓

instruction exception signal (red box). Figure 3.9 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[14]* is set to 0, the tag will be propagated from *gate 8* to *gate 10* and to *mux 12*. Then, since *mux 12* output drives one *gate 3* output, the exception is disabled. From Figure 3.9, seven fault propagation paths can be identified. We won't go into detail here about the seven different paths, as they were mentioned in case 2, bearing in mind that colour differentiation must be taken into account (for example: *alu_operand_a_ex_o_tag* instead of *store_dest_addr_ex_o_tag* from *gate 1* to *gate 3* if the fault is injected into *tcr_q[0]*, 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 into *rf_reg[14]*, and from *mux 11* to *gate 3* if a fault is injected into either *alu_operand_a_ex_o_tag* or *use_store_ops_ex_o*. Analysis of Figure 3.9 supports the results presented in Table 3.8 where *set to 0* and *bit-flip* fault types lead to successful attacks. As with first and second use cases the main reason is that the propagation paths are built entirely from *AND* gates.

3.4 Summary

In this section, I have described the processor, I will work on with its implementation of a hardware in-core DIFT. I have describe how it works and how to use the DIFT part with the default configuration. Then, I described the different use cases we choose to work with to analyse the DIFT behaviour and assess its behaviour against fault injection attacks. Finally, I presented the vulnerability assessment on these use case using the D-RI5CY processor. I have shown that this DIFT implementation is vulnerable to FIA in different registers and depending on the application a different path is used and so different registers will be criticals.

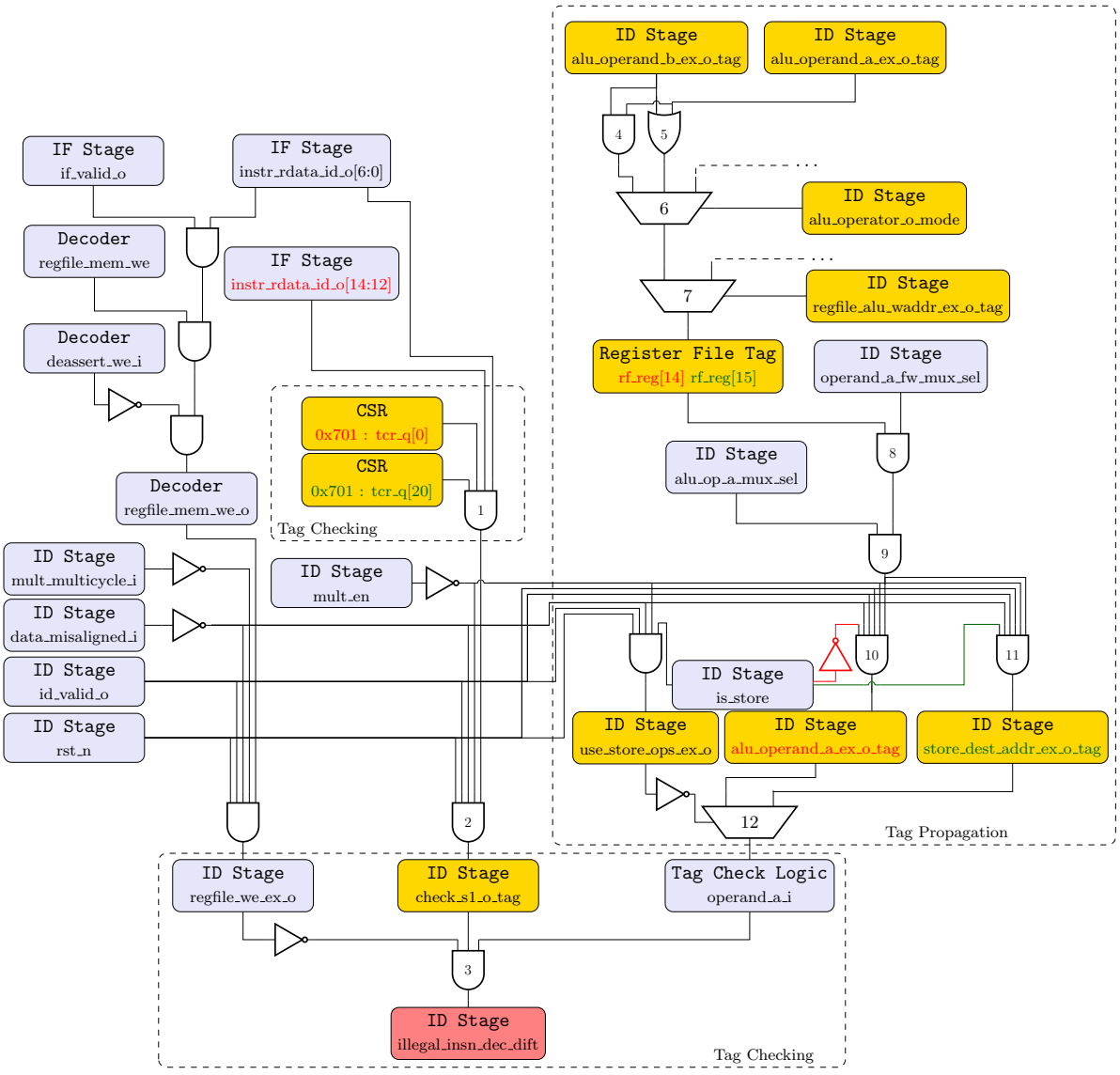


Figure 3.7: Logic description of the exception driving in a format string attack

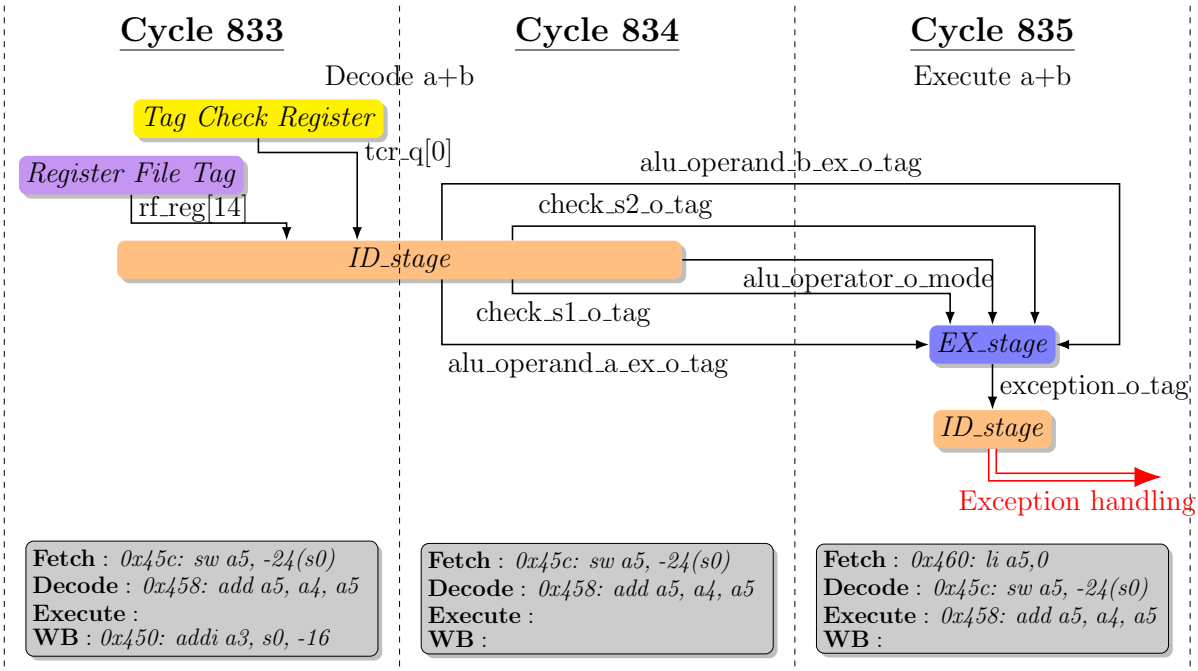


Figure 3.8: Tag propagation in a computation case with the compare/compute use case

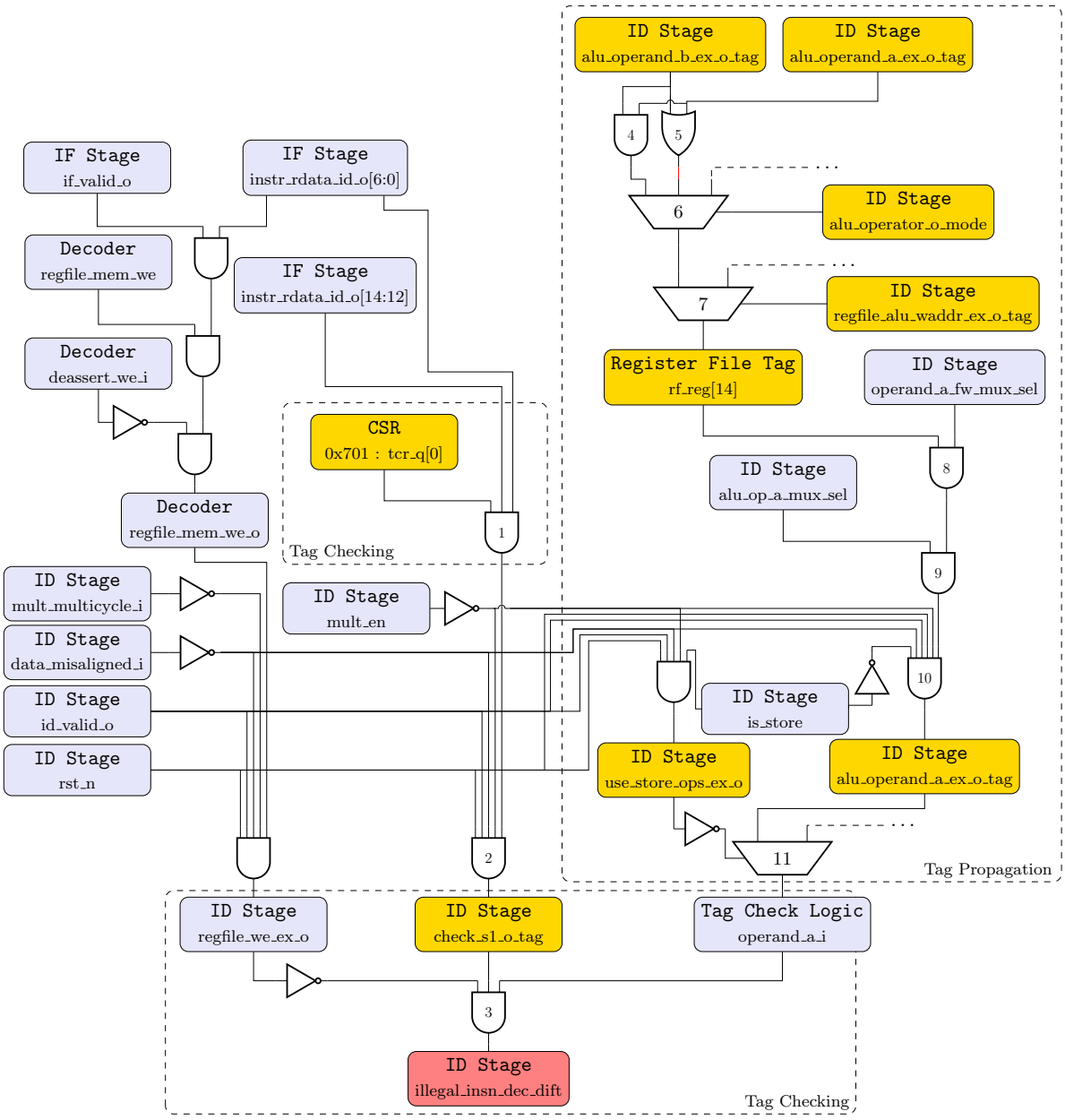


Figure 3.9: Logic representation of tag propagation in a computation case

Chapter 4

FISSA - Fault Injection Simulation for Security Assessment

Contents

This section will introduces and presents my tool, created to automate fault injection attacks campaigns in simulation. The first section will presents other tools for FIA campaigns in emulation, formal methods or even perform real world attacks. The second section will presents the architecure and details how FISSA works and presents how to extend it depending on other needs. Finally, I will discuss about it and draw some perspectives.

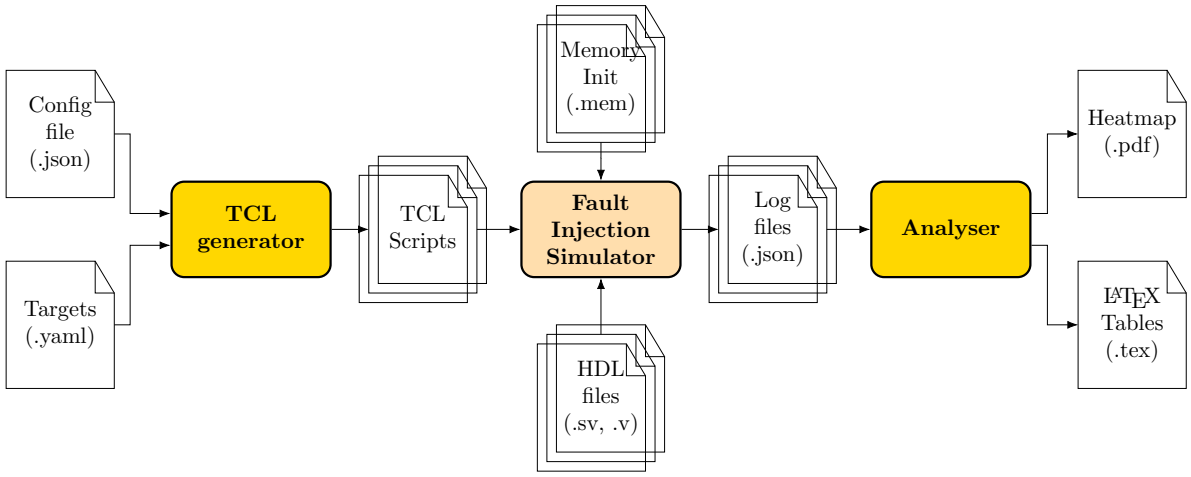


Figure 4.1: Software architecture of FISSA

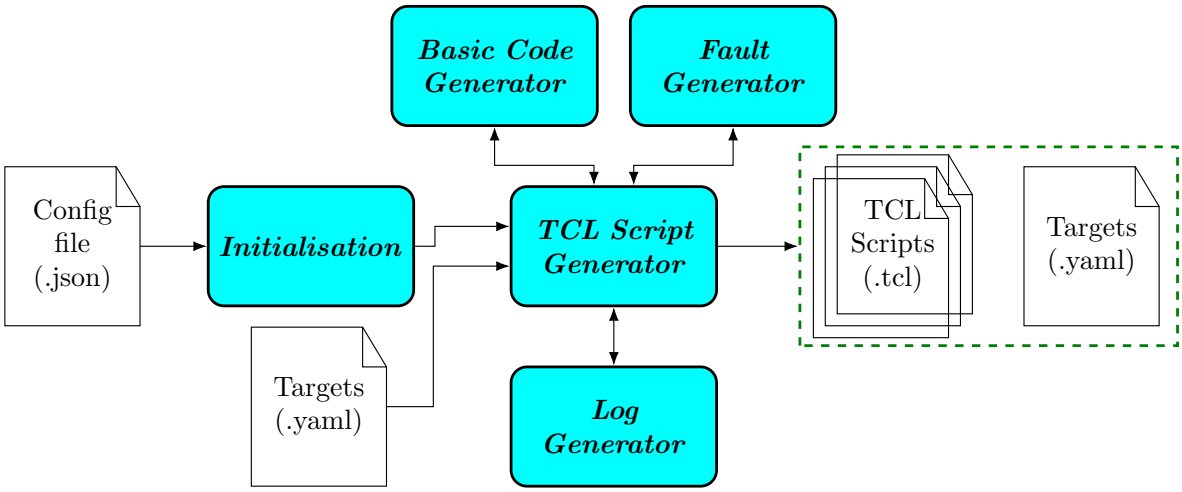


Figure 4.2: Software architecture of the TCL Generator module

4.1 Simulation tools for Fault Injection

4.2 FISSA

4.2.1 Main software architecture

4.2.2 Supported fault models

4.2.3 TCL Generator

4.2.4 Fault Injection Simulator

4.2.5 Analyser

4.2.6 Extending FISSA

4.3 Discussion and Perspectives

4.4 Summary

Chapter 5

Countermeasures Implementations

Contents

5.1 Countermeasure 1: Simple Parity

5.2 Countermeasure 2: Hamming Code

5.2.1 Implementation 1: Optimisation of redundancy bits

5.2.2 Implementation 2: Protection by pipeline stage

5.2.3 Implementation 3: Protection of all registers individually

5.2.4 Implementation 4: Protection of all registers individually with CSRs slicing

5.2.5 Implementation 5: Smart protection by pipeline stage

5.3 Countermeasure 3: Hamming Code - SECDED

5.3.1 Implementation 1: Optimisation of redundancy bits

5.3.2 Implementation 2: Protection by pipeline stage

5.3.3 Implementation 3: Protection of all registers individually

5.3.4 Implementation 4: Protection of all registers individually with CSRs slicing

5.3.5 Implementation 5: Smart protection by pipeline stage

5.4 Countermeasure 4: BCH Code

5.4.1 Implementation 1: 2 errors correction

5.4.2 Implementation 2: Generic implementation for n errors

5.5 Summary

Chapter 6

Experimental setup and results

Contents

6.1 Experimental setup

6.2 Experimental results

Chapter 7

Conclusion

The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards - and even then I have my doubts.

Gene Spafford

Contents

7.1 Synthesis

7.2 Perspectives

BIBLIOGRAPHY

[1] 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.

[2] 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.

[3] 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*, ser. SensorsS&P, Istanbul, Turkiye: Association for Computing Machinery, 2023, pp. 8–14, ISBN: 9798400704406. DOI: 10.1145/3628356.3630116.

Titre : titre (en français).....

Mot clés : de 3 à 6 mots clefs

Résumé : Eius populus ab incunabulis primis ad usque pueritiae tempus extremum, quod annis circumcluditur fere trecentis, circummurana pertulit bella, deinde aetatem ingressus adultam post multiplices bellorum aerumnas Alpes transcendit et fretum, in iuvenem erectus et virum ex omni plaga quam orbis ambit inmensus, reportavit laureas et triumphos, iamque vergens in senium et nomine solo aliquotiens vincens ad tranquilliora vitae discessit. Hoc inmaturo interitu ipse quoque sui pertaesus excessit e vita aetatis nono anno atque vicensimo cum quadriennio imperasset. natus apud Tuscos in Massa Veternensi, patre Constantio Constantini fratre imperatoris, matreque Galla. Thalassius vero ea tempestate praefectus praetorio praesens ipse quoque adrogantis ingenii, considerans incitationem eius ad multorum augeri discrimina, non maturitate vel consiliis mitigabat, ut aliquotiens celsae potestates iras principum molliverunt, sed adversando iurgandoque cum parum congrueret, eum ad rabiem potius evibrabat, Augustum actus eius exaggerando creberrime docens, idque, incertum qua mente, ne lateret adfectans. quibus mox Caesar acrius efferatus, velut contumaciae quoddam vexillum altius erigens, sine respectu salutis alienae vel suae ad vertenda opposita instar rapidi fluminis irrevocabili impetu ferebatur. Hae duae provinciae bello quondam piratico catervis mixtae praedonum.

Title: titre (en anglais).....

Keywords: de 3 à 6 mots clefs

Abstract: Eius populus ab incunabulis primis ad usque pueritiae tempus extremum, quod annis circumcluditur fere trecentis, circummurana pertulit bella, deinde aetatem ingressus adultam post multiplices bellorum aerumnas Alpes transcendit et fretum, in iuvenem erectus et virum ex omni plaga quam orbis ambit inmensus, reportavit laureas et triumphos, iamque vergens in senium et nomine solo aliquotiens vincens ad tranquilliora vitae discessit. Hoc inmaturo interitu ipse quoque sui pertaesus excessit e vita aetatis nono anno atque vicensimo cum quadriennio imperasset. natus apud Tuscos in Massa Veternensi, patre Constantio Constantini fratre imperatoris, matreque Galla. Thalassius vero ea tempestate praefectus praetorio praesens ipse quoque adrogantis ingenii, considerans incitationem eius ad multorum augeri discrimina, non maturitate vel consiliis mitigabat, ut aliquotiens celsae potestates iras principum molliverunt, sed adversando iurgandoque cum parum congrueret, eum ad rabiem potius evibrabat, Augustum actus eius exaggerando creberrime docens, idque, incertum qua mente, ne lateret adfectans. quibus mox Caesar acrius efferatus, velut contumaciae quoddam vexillum altius erigens, sine respectu salutis alienae vel suae ad vertenda opposita instar rapidi fluminis irrevocabili impetu ferebatur. Hae duae provinciae bello quondam piratico catervis mixtae praedonum.