THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE BRETAGNE SUD

ÉCOLE DOCTORALE Nº 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 Nº : « 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

$\mathscr{A}$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

API	Application Programming Interface
CABA	Cycle Accurate and Bit Accurate
CSR	Control and Status Registers
DIFT	Dynamic Information Flow Tracking
FIA	Fault Injection Attack
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
ISA	Instruction Set Architecture
OS	Operating System
PC	Program Counter
RA	Return Address
SCA	Side Channel Attack
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, 2]

1.3 Objectives

1.4 Manuscript outline

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

Chapter 2 presents the state of the art about Information Flow Tracking (IFT) by explaining how they work, the different types of existing IFT, this chapter presents what is physical attacks, the literature about it and presents the two principle type of physical attacks: Side-Channel Attacks and Fault Injection Attacks. Finally, the chapter presents an overview of the literature about countermeasures against Fault Injection Attacks

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, FISSA, to automatise fault injection campaigns in simulation. This tool allows a designer to assess his design during the conception phase. This chapter presents how it works and how to use it, and compares it to others tools available in the literature.

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, in Section 2.2, Information Flow Tracking (IFT) is introduced, detailing the different types and their respective purposes. We will discuss the various levels of monitoring, from program behaviour to the detection of hardware trojans. Then in Section 2.3, Physical Attacks are examined, focusing on two main types: Side-Channel Attacks (SCA) and Fault Injection Attacks (FIA). Finally in Section 2.4, as this work will concentrate on FIA, we will exclusively present countermeasures against Fault Injection Attacks.

2.2 Information Flow Tracking

The concept of *Information Flow Tracking* has been introduced by the work of Bell and LaPadula [3] and by Denning [4] in 1976. This section introduces Information Flow Tracking mechanisms, explains how they work, and presents the various types of IFT with their different functional levels.

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

Static Information Flow Tracking (SIFT) is a security technique used to analyse and control the flow of information within a program or system without executing it, by examining the source code or compiled binary [5]. This method is particularly useful for identifying theoretical vulnerabilities, ensuring compliance with design principles, and preventing unauthorised information leaks before deployment. SIFT is comprehensive, covering all possible execution paths and detecting both explicit information flows (direct data assignments) and implicit flows (leaks through control flow structures). By performing checks at compile-time, SIFT helps developers address potential security issues early, enforcing principles like non-interference and data confidentiality through security policies. However, static analysis may generate false positives by flagging theoretical flows that might not occur in practice and may struggle with certain dynamic language features or runtime-dependent behaviours. SIFT is employed in various contexts, such as verifying secure information flow in operating systems, programming languages with built-in information flow controls, and hardware design for secure systems.

2.2.1.2 Dynamic IFT

Dynamic Information Flow Tracking (DIFT) is a powerful security technique that monitors and analyses, in real-time, the flow of information within a program during its execution [6]. DIFT operates by tagging or labelling input data from potentially untrusted sources and tracking how this data propagates through the system [7]. As the program executes, DIFT maintains metadata about the tagged information, updating it as operations are performed on the data. This allows the system to detect when tainted data is used in security-critical operations, such as modifying control flow or accessing sensitive resources. DIFT can be implemented at various levels, including hardware, software, or a combination of both. Hardware-based implementations often offer better performance but require specialized processor modifications, while software-based approaches provide more flexibility but may incur higher overhead [6]. DIFT has proven effective in detecting and preventing a wide range of security vulnerabilities, including buffer overflows, format string attacks, and code injection attacks [7]. However, DIFT also faces challenges, such as handling implicit information flows, managing performance overhead, and addressing over-tainting issues. This approach might not cover all potential data paths, as it is dependent on the specific conditions and inputs provided during the monitoring period. Despite these challenges, DIFT remains a valuable tool for software security, particularly for runtime attack detection in modern systems.

2.2.2 Different levels of DIFT

IFT can be implemented at various levels of abstraction in computing systems [5, 6, 8]. Each level presents unique trade-offs between precision, performance overhead, and ease of implementation, allowing designers to choose the most appropriate approach for their security requirements.

Software-based DIFT mechanisms benefit from close integration with the software context via binary code instrumentation and source code modifications, offering better flexibility, customisation, and scalability without altering hardware components. However, these software solutions often incur high performance overheads due to the extra instructions required. They operate at either the system level, monitoring OS-wide information flows, or the program level, focusing on specific applications. On the other hand, hardware-assisted DIFT designs can efficiently enforce security rules by implementing DIFT-related operations as hardware logic, reducing performance overhead but at the expense of flexibility and scalability, making them challenging to deploy in modern commercial systems. They can be implemented within processor cores or as off-core designs. But they can also be at the lowest level such as Gate-Level IFT who tracks information flow through logic gates. A hybrid hardware and software co-design offers a promising alternative, enabling fine-grained security checks by associating software context with hardware data, though it faces challenges such as balancing flexibility with hardware overhead and designing appropriate tags that support rule updates post-deployment.

Figure 2.1 represents the different levels of a simplified embedded system: application layer, system service layer, OS layer, and hardware layer. This figure is inspired by Figure 1.9 of [9]. Software-based IFTs work in the first three levels.

Positioned at the highest level of the software hierarchy, *the application layer* is responsible for implementing system functionalities and business logic. Functionally, all modules within this layer work together to execute the required system operations. Applications generally run in a less-privileged mode on the processor and utilise the OS-provided API scheduling to communicate with the operating system. *The system service layer* serves as the intermediary service interface offered by the OS to the application layer. This interface allows applications to access a variety of OS-provided services, essentially bridging the gap between the OS and applications. Typically, this layer encompasses components like the file system, Graphical User Interface (GUI), task manager. An Operating System (OS) is a software framework designed to manage hardware resources uniformly. It abstracts numerous hardware functions and offers them to applications as services. Common services provided by an OS include scheduling, file synchronisation, and networking. Operating systems are prevalent in both desktop and embedded systems. In the context of embedded systems, OSs possess distinct characteristics such as stability, customisability, modularity, and real-time processing capabilities. *The hardware layer* refers to the physical components and circuitry, including the microprocessor or microcontroller, memory, sensors, and input/output interfaces. This layer encompasses all the tangible electronic elements that interact directly with each other to perform the device's functions. It provides the essential infrastructure that supports and drives the embedded system's operations and connectivity.

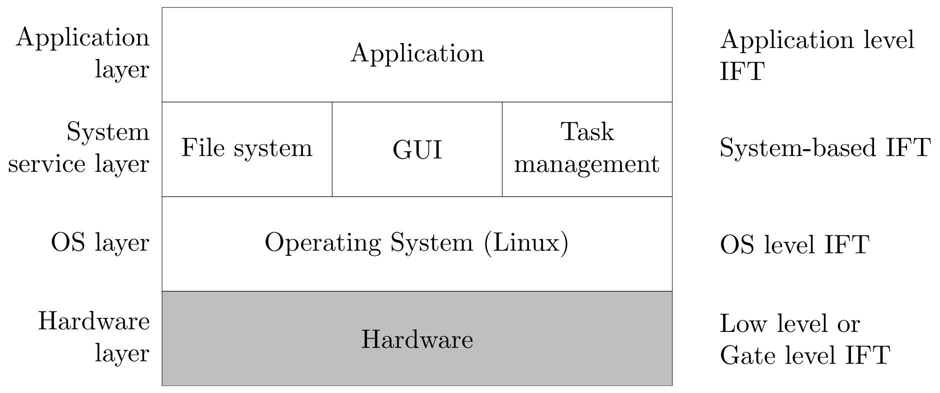


Figure 2.1: Simplified representation of the different layers in an embedded system

Tracking information can be performed at various levels, from the application level to the hardware level. Each level offers distinct advantages and disadvantages. For instance, application-level tracking might provide detailed insights and user-friendly interfaces, while hardware-level tracking offers more granular data and real-time monitoring but can be more complex and costly. The following subsections will explore these different levels, highlighting their respective benefits and limitations.

2.2.2.1 Software-based DIFT

Application level DIFT tracks information flows between application variables. The programmer has to integrate data tagging inside his program and use a modified compiler or analyse his program to check if no security violation happened. One application for DIFT at application level is language-based. Several security extensions have been proposed for existing programming languages. JFlow [10] is one of the first works that has described an extension of the Java language by adding statically-checked information flow annotations.

Multiples works introduce DIFT extensions for different languages, for example such as JavaScript [11, 12]. Austin et al. [12] propose a method for tracking information flow in dynamically-typed languages, focusing on addressing issues with implicit paths through a dynamic check. This approach avoids the necessity for approximate static analyses while still ensuring non-interference. The method employs sparse information labelling, keeping flow labels implicit where possible and introducing explicit labels only for values crossing security domains.

Kemerlis et al. [13] provide a framework, *libdft*, which is fast and reusable and applicable to software and hardware. *libdft* provides an API for building DFT-enabled tools that work on unmodified binaries.

OS level and System-based DIFT track and tag files (read or written) used by the application. The main advantage of this approach is that it reduces the number of information flows which lead to an improvement of the runtime overhead. In the other side, the main disadvantage of this approach is that it results in more false positives than the application-level approach.

TaintDroid [14] introduces an extension to the Android mobile phone platform designed to monitor the flow of privacy-sensitive data through third-party applications. Operating under the assumption that downloaded third-party applications are untrusted, TaintDroid tracks in real-time how these applications access and handle users' personal information. The primary objectives are to detect when sensitive data is transmitted out of the system by untrusted applications and to enable phone users or external security services to analyse these applications. They store the data adjacent to data for spatial locality. This may cause large performance and storage overheads, as the tag fetching requires extra clock cycles for memory access. HiStar [15] is a new OS that has been designed to provide precise data specific security policies. The authors made the choice to assign tags to different objects in the operating system instead of data.

2.2.2.2 Software and Hardware Co-Design-Based DIFT

This type of design combines the features of both software DIFT and hardware DIFT. Using binary instrumentations and a modified compiler, the hardware and software co-design can provide the best of these two categories of DIFT: flexible security configuration and fine-grained protection with low impact on performances [6, 8].

One example of this type of DIFT is RIFLE [16], a runtime information-flow security system designed from the user's perspective, provides a practical means to enforce information-flow security policies on all programs by leveraging architectural support. RIFLE employs binary instrumentation and architectural support to enforce information flow security policies during runtime. The conventional Instruction Set Architecture (ISA) is transformed into an Information-Flow Security (IFS) ISA using a dedicated binary translator. Each instruction in the ISA corresponds to an instruction in the IFS ISA. In the IFS ISA, additional security registers are assigned to hold tags. To avoid the pitfalls dynamic mechanisms encountered while tracking implicit flows, the binary translation will convert all implicit flows to explicit flows. The RIFLE architecture is then responsible only for tracking explicit flows. The translated binary is executed within a modified processor supporting the IFS ISA, which is simulated within the Liberty Simulation Environment. RIFLE works with every programs that run on a system and policy decisions are left to the user, not the programmer.

Townley et al. [17] presented LATCH, a generalizable architecture for optimizing DIFT. LATCH exploits the observation that information flows under DIFT exhibit strong spatial and temporal locality, with typical applications manipulating sensitive data during limited phases of computation. The main objective is to detect attacks on the integrity of the system. The architecture consists of a software-assisted hardware accelerator (S-LATCH) running on a single simulated core. The software component of S-LATCH propagates tags, while the hardware accelerator monitors the data accessed by the program to detect tags.

Porquet et al. [18] presented WHISK, a whole system DIFT architecture implemented within a hardware simulator. WHISK stores tags and data separately in memory locations to keep low area overhead and improve flexibility and to better accommodate the integration of hardware accelerators.. Tag insertion, storage and access to the custom hardware are delegated. The software subsystem uses MutekH exokernel-based OS and provides support for tag page allocation, page table cache configuration, and interrupt handling concerning writes to untagged pages.

2.2.2.3 Hardware-based DIFT

Dalton et al. [19] report that software DIFT solutions add significative runtime overhead, up to a slow down of 37 times ! Therefore, in order to improve the execution time to be more on-the-fly, the idea is to directly implement the DIFT into the hardware, but the trade-off is flexibility. This subsection discusses the hardware-based DIFT designs including gate-level DIFT designs and micro-architecture-level DIFT designs. Surveys [5, 8] present an overview on all hardware DIFT techniques. They developed a taxonomy for them and use it to classify and differentiate hardware DIFT tools and techniques.

Gate-Level DIFT include gate-level netlist and also RTL designs. The goal is to protect against hardware trojans and unauthorized behaviors. To achieve that, during the creation of the circuit, additional logic is added for each gate used in the design.

GLIFT [20] is a well-established IFT technique. The goal is to protect against hardware trojans and unauthorized behaviors. All information flows, both explicit and implicit, are unified at the gate level. GLIFT employs a detailed initialisation and propagation policy to precisely track each bit of information flow, by adding additional logic for each gate used in the design. By analysing how inputs influence outputs, GLIFT accurately measures true information flows and substantially reduces the false positives typically associated with conservative IFT techniques. Hu et al. [21] established the theoretical foundation for GLIFT. They introduced several algorithms for generating GLIFT logic in large digital circuits. Additionally, the authors identified the primary source of precision discrepancies in GLIFT logic produced by various methods as static logic hazards or variable correlation due to reconvergent fanouts. Many other works have been done on GLIFT to attempt a decrease of the logic complexity.

Off-Core DIFT operations are performed on a dedicated coprocessor working in parallel of the main core. The main drawback is that this approach needs a support from the OS for the synchronisation between data computations and tags computations in order to stall one core if it needs to wait the other. But on the other hand, its advantage is that it does not require internal hardware modifications to the main kernel. Processor manufacturers do not prioritise this type of security, and as most processors are not open to the public, it is difficult to modify them.

Kannan et al. [22] described one of the first work using a coprocessor to improve tag computation runtime overhead. Traditional hardware DIFT systems require significant modifications to the processor pipeline, which increases complexity and design time. Figure 2.2 represent how an off-core DIFT would be implemented. Kannan et al. uses this idea for implementing their solution of DIFT. This coprocessor handles all DIFT functionalities, synchronizing with the main processor only during system calls. This design eliminates the need for changes to the processor's pipeline, logic, or caches, making it more attractive. The coprocessor is small, with an area footprint of about 8% of a simple RISC core, and introduces less than 1% runtime overhead for SPECint2000 applications benchmark. The paper demonstrates that the coprocessor provides the same security guarantees as integrated DIFT architectures, supporting multiple security policies and protecting various memory regions and binary types. This approach offers a balanced solution in terms of performance, cost, complexity, and practicality compared to existing DIFT implementations.

Wahab et al. [23, 24] developped a DIFT using the ARM CoreSight debug component to extract a trace. However, the debug component could only extract limited information about the application executing on the core. Therefore, some instrumentations has been required to recover the complete program trace. The information obtained from the trace is then sent to a dedicated DIFT coprocessor, which analyses the instruction trace and propagates tags according to a security policy. In terms of performance and area footprint, [24] gives around 5% communication overhead and an area overhead of 0.47% and a power consumption increased by 16%; while [23] gives a communication overhead of 335%, an area increased by 0.95% and a power consumption increased by 16.2%. These results can not be compared to the initial design as they use a coprocessor without the ARM core results.

Off-Loading DIFT use a dedicated core of a multi-core CPU [25–27]. Figure 2.3 represents how Off-Loading DIFT works with a core running the application and another, in parallel, run the DIFT analysis on the application trace. The application core is modified to create a trace and compress it. The trace includes executed instructions and pack main information such as PC address, register operands. This trace is then sent to the DIFT core via the L2 cache. Finally, the security core will decompress the trace and realizes tag computation in order to check whether an illegal information flow has been done.

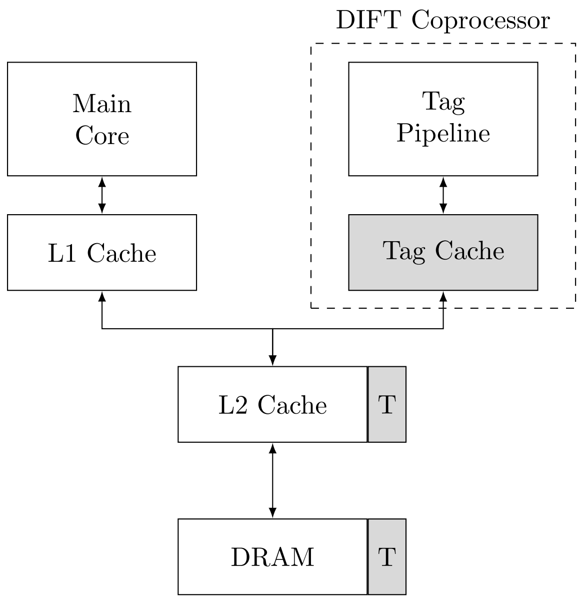


Figure 2.2: Representation of an Hardware Off-Core DIFT (inspired by Figure 1 of [22])

The notion of illegal information flow is specified thanks to a DIFT security policy. The main advantage is that hardware does not need to know DIFT tags or policies and does not need a coprocessor with the management of the synchronisation between the two processors. But the main drawback is that it requires a multi-core CPU but reduces the number of core available and doubles the energy consumption due to the application trace analysis. In an embedded systems where consumption is a critical factor this solution is difficult to consider.

In-Core DIFT rely on a deeply modified processor pipeline which need to integrate tag computations inside the main core in parallel of data computations. This approach is highly invasive, but does not require any additional cores or coprocessors to operate and introduces no overhead for inter-core synchronisation. Overall, its performance impact in terms of clock cycles over native execution is minimal. On the other hand, the integrated approach requires significant modifications to the processor core. All pipeline stages must be modified to add tags, a dedicated register file and first level of caches must be added to store tags in parallel with the regular blocks into the processor core. Figure 2.4 shows the architecture of an In-Core hardware DIFT. When the processor fetches an instruction, its associated tag is sent in parallel. In the decode stage, the instruction is decoded while the security decode module decode the security policy to determine how the tag should be propagated and checked. When the instruction is executed, the tag is sent to a tag ALU to be checked. Then, if the tag is conform to the security policy the tag and the ALU output will be saved into the Data-Cache to be used again or stored in memory. Otherwise, if the tag is not conform, the DIFT mechanism will detect the security violation and can raise an exception, stop the application, depending on what is configured.

Suh et al. [7] proposed an approach in which the OS identifies a set of input channels as spurious, and the processor tracks all information flows from these inputs. Thanks to this tracking, the processor

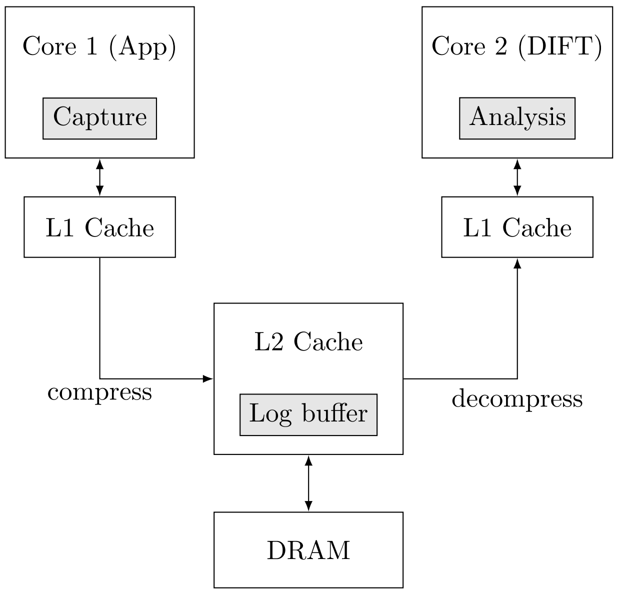


Figure 2.3: Representation of an Hardware Off-Loading DIFT (inspired by Figure 1 of [22])

can detect various attacks such as attack targetting instructions or jump addresses. If the security policy detects something malicious in hardware, the OS will process the exception. They use 1-bit tag which means only two ways of representig security levels. They present two security policies that track differing sets of dependencies. Implementing the first policy incurs, on average, a memory overhead of 0.26% and a performance decrease of 0.02%. The second policy incurs, on average, a memory overhead of 4.48% and a performance decrease of 0.8%, and requires binary annotation unlike first policy.

Dalton et al.[19] presented a DIFT architecture, Raksha, to support a flexible security configuration at runtime. They extended all storage locations including registers, caches and main memory with tags, they modified the ISA instruction to propagate and check tags. In this solution, they use 4-bits tags for each word. These tags represent the security policy and not the data state (e.g. trusted or untrusted). The authors provided two global sets of configuration registers, i.e., Tag Propagation Registers (TPR) and Tag Check Registers (TCR), to configure the security policy at runtime. There is one pair of TPR/TCR for each of the four security policies. The configuration register could be configured only in trusted mode. Moreover, the tag propagation and check could only be disabled in trusted mode. However, the security policy is difficult to update when the architecture is deployed. The Raksha prototype is based on the Leon SPARC V8 processor, a 32-bit open-source synthesizable core, and they mapped the design onto a FPGA board.

Palmiero et al. [28] implemented a DIFT framework, D-RI5CY, on a RISC-V processor and synthesized it on a Field Programmable Gate Array (FPGA) board with a focus on IoT applications. The proposed design tags every word in data memory with a 4-bits tag and every general register with a 1-bit tag. Similarly to [19], Palmiero et al. [28] also adopted global configuration registers to customise the rule of tag propagation and checking. Each type of instruction has its own rule and can be modified separately. This method provides a more fine-grain tracking than Raksha but lacks flexibility for security policy reconfiguration for different program contexts.

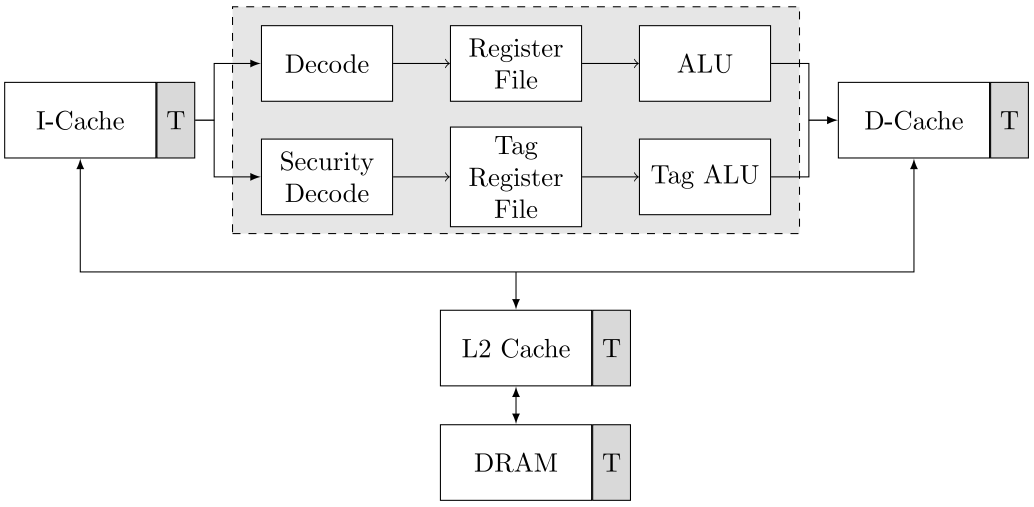


Figure 2.4: Representation of an Hardware In-Core DIFT (inspired by Figure 1 of [22])

2.2.3 How DIFT works

William ►*expliquer DIFT ici ? en détail avec les schémas explicatifs*◄

2.3 Physical Attacks

2.3.1 Side-Channel Attacks

2.3.2 Fault Injection Attacks

[29]

2.4 Countermeasures against FIA

Chapter 3

D-RI5CY - Vulnerability 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 DIFT design under consideration. The second section details and describes the considered uses cases of this thesis. Finally, the third section assesses the vulnerabilities of the D-RI5CY, using these three cases.

3.1 D-RI5CY

In this section, we describe the RISC-V ISA and detail the DIFT design we have chosen to focus on. We choose to work on RISC-V core as they are open-source, and it means that we have the ability to access and modify the design according to our needs.

3.1.1 RISC-V Instruction Set Architecture (ISA)

RISC-V is an open and free ISA, which was originally developed at University of California, Berkeley, in 2010, and now is managed and supported by the RISC-V Foundation, having more than 70 members

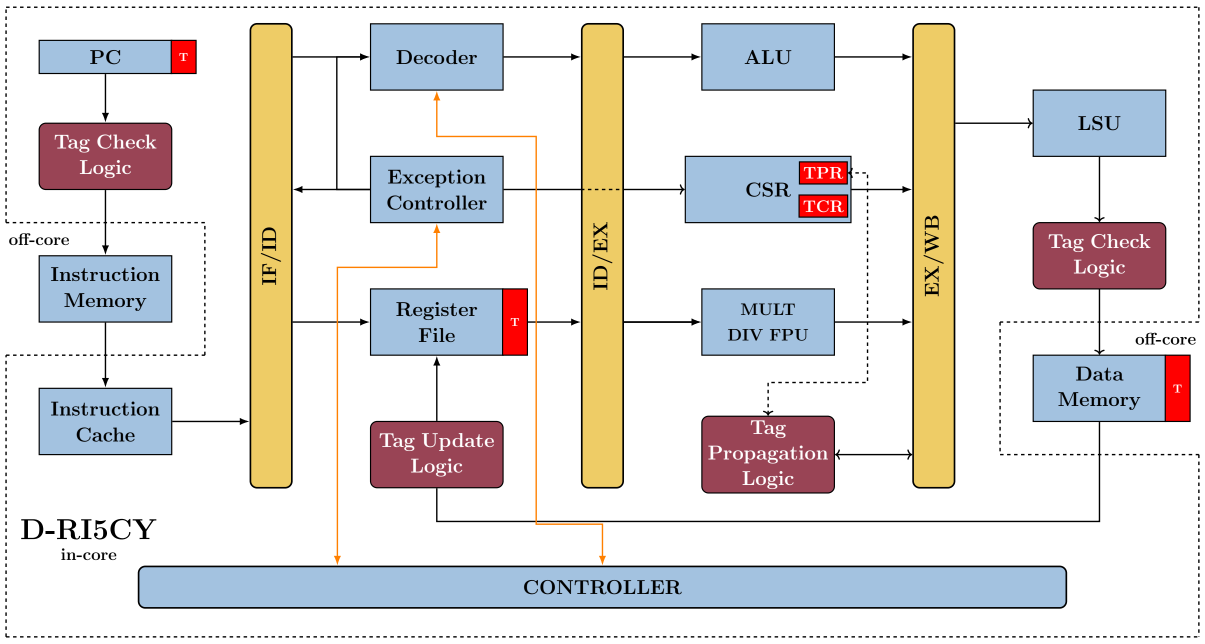


Figure 3.1: D-RI5CY processor architecture overview. DIFT-related modules are highlighted in red.

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 ISA, RISC-V is freely available for anyone to 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.

3.1.2 DIFT design

This thesis focus on the evaluation of a DIFT against fault injection attacks in order to protect it. We opted to not develop a Dynamic Information Flow Tracking (DIFT) system from scratch, as this would have required considerable time for implementation and testing, which was not within the scope of our objectives. Consequently, we decided to review the current state of the art and select an open-source DIFT system. As a result, we have selected the D-RI5CY [28, 30] 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 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 and dark red are represented the DIFT specific modules. These modules allow tags to be initialised, propagated and checked during the execution 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 in parallel to its associated data, the *Tag Propagation Logic* module propagates it 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 and the application is stopped to avoid accessing or executing corrupted data.

The authors of the D-RI5CY defined a library of routines to initialise the tags of the data coming from potentially malicious channels. At program startup, D-RI5CY initialises the tags of the registers, program counter and memory blocks to *zero*. 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. They extended the RI5CY ISA with memory and register tagging instructions. They have added four assembly instructions to initialise tags for user-supplied inputs:

- **p.set rd**: sets to untrusted the security tag of the destination register *rd* (you can check the register names in the ISA specification[1] at page 85),
- **p.spsb x0, offset(rt)**: sets to untrusted the security tag of the memory byte at the address of the value stored in *rt + offset*,
- **p.spsh x0, offset(rt)**: sets to untrusted the security tag of the memory half-word at the address of the value stored in *rt + offset*,
- **p.spsw x0, offset(rt)**: sets to untrusted the security tag of the memory word at the address of the value stored in *rt + offset.*

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 in Figure 3.1). Finally, they added 4-bit tags to the data memory. 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 additional 32-bits registers : Tag Propagation Register (TPR) and Tag Check Register (TCR). These registers are used to store the security policy for both tag propagation and tag check. These registers contain a default policy, and they can be modified during runtime with a simple *csr write* instruction, such as `csrw csr, rs1`. These policies

1. `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf`

Table 3.1: Instructions per category

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]*

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

consist of rules, which have fine-grain control over tag propagation and tag 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. 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 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. The tag control policy determines whether 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 instructions. 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 summarise, at first ①, D-RI5CY initialises the configuration registers (TPR and TCR) from the

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	1 0 1 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0	0 0 0
Policy 2	0	0 0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0	0 1 1

default security policy. Then at program startup ②, D-RI5CY initialises all the tags to *trusted* (i.e, set to 0). The tag propagation ③ and verification ④ happen in the D-RI5CY pipeline in parallel with the standard behaviour, without incurring any latency overhead.

3.1.3 Pedagogical case study

To present the use of the D-RISCY, we will introduce a use case to demonstrate how to use a new security policy and how the DIFT will detect the violation of different security policies. This use case has been developed for pedagogical purposes but does not involve a real software attack.

Listing 3.1 shows the C code used for this use case. Lines 2 to 4 initialize variables, lines 5 and 6 configure a security policy by writing in the TPR and TCR registers thanks to an assembly line. Line 7 tags the variable "`a`" as untrusted (tag is set to "*1*"). 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`". 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.

The assembly line in C is constructed from key words *asm volatile*. The template for this assembly line is: "*asm asm-qualifiers (AssemblerTemplate : OutputOperands [: InputOperands [: Clobbers]])*". So to explain briefly, line 7 in Listing 3.1 is composed of a custom assembly instruction "`p.spsw`", that takes the "`x0`" register as target and specifies an address mode using the placeholder "`0(%0)`". Finally, "*:: "r" (&a)*" part specifies the input operand, with "`r`" indicating that a general-purpose register should be used to hold the address of the variable "`a`".

In terms of security policy, depending on which one we use in Table 3.2 and Table 3.3, we will have different results of exception. Security policy 1 propagates the tags with an *OR* logic for five modes (arithmetic, jump, shift, logical, and load/store mode) and enables the propagation of the tag from the source of a load/store. Security policy 1 checks the tags only for the execute check (i.e., PC instruction) and for the source address and destination address for a load/store instruction. In comparison, security policy 2 enables the propagation for all tags and checks tags only for both inputs of arithmetic instructions. To summarise from our application case, if we use security policy 1, the DIFT will detect the *load* instruction before executing the "`a > b`" comparison and raise an exception; whereas if we use security policy 2, the DIFT protection raises an exception when executing the instruction `add a5,a4,a5` (i.e., the "`a+b`" C statement), since variable `a` is untrusted and `b > a`.

In the continuation of this work, this use case will be referred to as *Compare/Compute* and will be utilised as the third case, implementing security policy 2 from Table 3.2 and Table 3.3. The two other use cases will be presented in the following section 3.2.

Listing 3.1: 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;
}
```

3.2 Use cases

This section details the considered use cases in our work. The first two use cases come from the original paper [28]. 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[2] 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.

To illustrate the use of TCR and TPR registers, 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).

Listing 3.2 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.

2. `https://github.com/sld-columbia/riscv-dift/blob/master/pulpino_apps_dift/wilander_testbed/`

Figure 3.2 represents the five steps from the source buffer initialisation to the first shellcode instruction being fetched. In Figure 3.2a, the source buffer, in yellow, is initialised with A's, and as it is manipulated by a user, it is tagged as untrusted (red). The destination buffer is empty, and both *PC* and *RA* register are trusted (green). In Figure 3.2b, the source buffer is copied into the destination buffer, the data and the tag are copied. In Figure 3.2c, the overflow occurs and the *ra* register is compromised with the address of the shellcode function from the source buffer. Now, all the memory tags are untrusted. In Figure 3.2d, the *PC* loads the *ra* register along with its tag. The *PC* loses its integrity and became untrusted. In Figure 3.2e, the *PC* address is fetched, and the instruction is sent into the pipeline along with the tag. At this moment, the DIFT mechanism will detect the untrusted tag and as the security policy do not allow executing an untrusted PC, an exception will be raised and the application will be stopped.

Listing 3.2: Buffer overflow C code

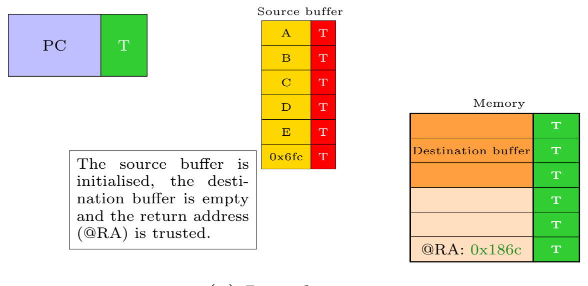
```c
#define BUFSIZE 16
#define OVERFLOWSIZE 256

int base_pointer_offset;
long overflow_buffer[OVERFLOWSIZE];

int shellcode() {
    printf("Success !!\n");
    exit(0);
}

void vuln_stack_return_addr(){
    long *stack_pointer;
    long stack_buffer[BUFSIZE];
    char propolice_dummy[10];
    int overflow;

    /* Just a dummy pointer setup */
    stack_pointer = &stack_buffer[1];

    /* Store in i the address of the stack frame section dedicated to function arguments */
    register int i asm("x8");

    /* First set up overflow_buffer with 'A's and a new return address */
    overflow = (int)((long)i - (long)&stack_buffer);
    memset(overflow_buffer, 'A', overflow-4);
    overflow_buffer[overflow/4-1] = (long)&shellcode;

    /* TAG INITIALISATION */
    for(int j=0; j<overflow/4; j++) {
        asm volatile ("p.spsw x0, 0(%[ovf]);"
                     ::[ovf] "r" (overflow_buffer+j));
    }

    /* Then overflow stack_buffer with overflow_buffer */
    memcpy(stack_buffer, overflow_buffer, overflow);

    return;
}

int main(){
    vuln_stack_return_addr();
    printf("Attack prevented.\n");
    return EXIT_SUCCESS;
}
```

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

The second use case is a format string attack[3] overwriting the return address of a function to jump to a shellcode and starts its execution. This use case uses the first security policy from Table 3.2 and Table 3.3. 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 [31] for detailed information) to write the targeted address.

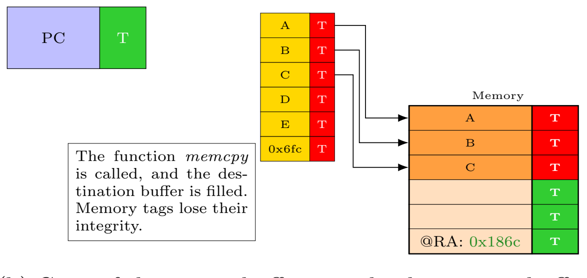
Listing 3.3 shows the C code of this use case. The `echo` function assign the *x8* register to a variable '`i`' which goes into another variable '`a`'. The lines 13-14 are used to initialise the tag associated to the

3. `https://github.com/sld-columbia/riscv-dift/tree/master/pulpino_apps_dift/wu-ftpd`



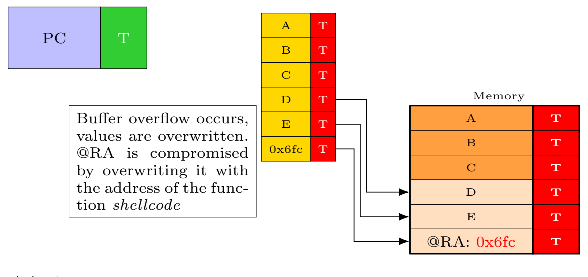
PC
T
Source buffer
A
B
C
D
E
0x6fc
T
Memory
Destination buffer
@RA: 0x186c
The source buffer is initialised, the destination buffer is empty and the return address (@RA) is trusted.

(a) Initialisation



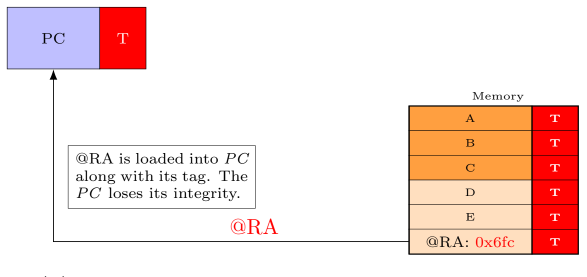
PC
T
A
B
C
D
E
0x6fc
Memory
@RA: 0x186c
The function *memcpy* is called, and the destination buffer is filled. Memory tags lose their integrity.

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



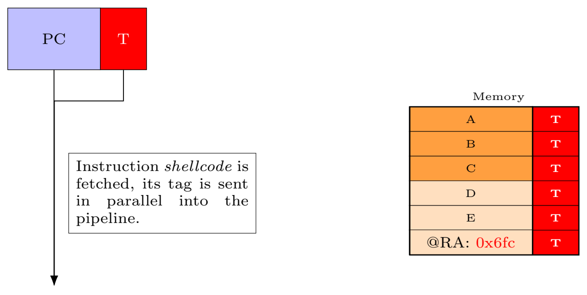
PC
T
A
B
C
D
E
0x6fc
Memory
@RA: 0x6fc
Buffer overflow occurs, values are overwritten. @RA is compromised by overwriting it with the address of the function *shellcode*

(c) An overflow occurs, the *ra* register is overwritten



PC
T
Memory
@RA: 0x6fc
@RA is loaded into *PC* along with its tag. The *PC* loses its integrity.
@RA

(d) Corrupted *ra* register is loaded into the PC



PC
T
Instruction *shellcode* is fetched, its tag is sent in parallel into the pipeline.
Memory
A
B
C
D
E
@RA: 0x6fc

(e) PC address instruction is fetched

Figure 3.2: Representation of how the ROP attack works

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 3.3. 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

Listing 3.3: WU-FTPd C code

```c
void secretFunction(){
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");

    exit(0);
}

void echo(){
    int a;
    register int i asm("x8");
    a = i;

    asm volatile ("p.spsw x0, 0(%[a]);"
                  ::[a] "r" (&a));

    printf("%224u%n%35u%n%253u%n%n", 1, (int*) (a-4), 1, (int*) (a-3), 1, (int*) (a-2), (int*) (a-1));

    return;
}

int main(int argc, char* argv[]){
    volatile int a = 1;

    if(a)
        echo();
    else
        secretFunction();

    return 0;
}
```

Table 3.4: Memory overwrite

Address	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

our architecture, the address fetched by the pipeline is only '*000003E0*'.

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. For these campaigns, we use a tool, developed for this purpose. This tool is described in Chapter 4 and can generate the TCL code to automatise fault injections attacks campaigns at Cycle Accurate and Bit Accurate (CABA) level. 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 ACM Sensors S&P [32].

We assess the design with fault injection campaigns. With their results associated, we can deduce which registers are vulnerable 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.

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**

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	bit-flip	set0	set1	bit-flip	set0	set1	bit-flip	set0	set1	bit-flip	set0	set1	bit-flip
pc_if_o_tag										✓		✓			
rf_reg[1]							✓		✓						
tcr_q	✓			✓			✓			✓			✓		
tcr_q[21]			✓			✓			✓			✓			✓
tpr_q	✓	✓		✓	✓										
tpr_q[12]			✓			✓									
tpr_q[15]			✓			✓									

3.3.1 Fault model for vulnerability assessment

In this vulnerability assessment, we consider an attacker able to inject faults into DIFT-related registers leading to *set to 0*, *set to 1*, and *single bit-flip in one register at a given clock cycle.* To bypass the DIFT mechanism, the main attacker's goal is to prevent an exception 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.

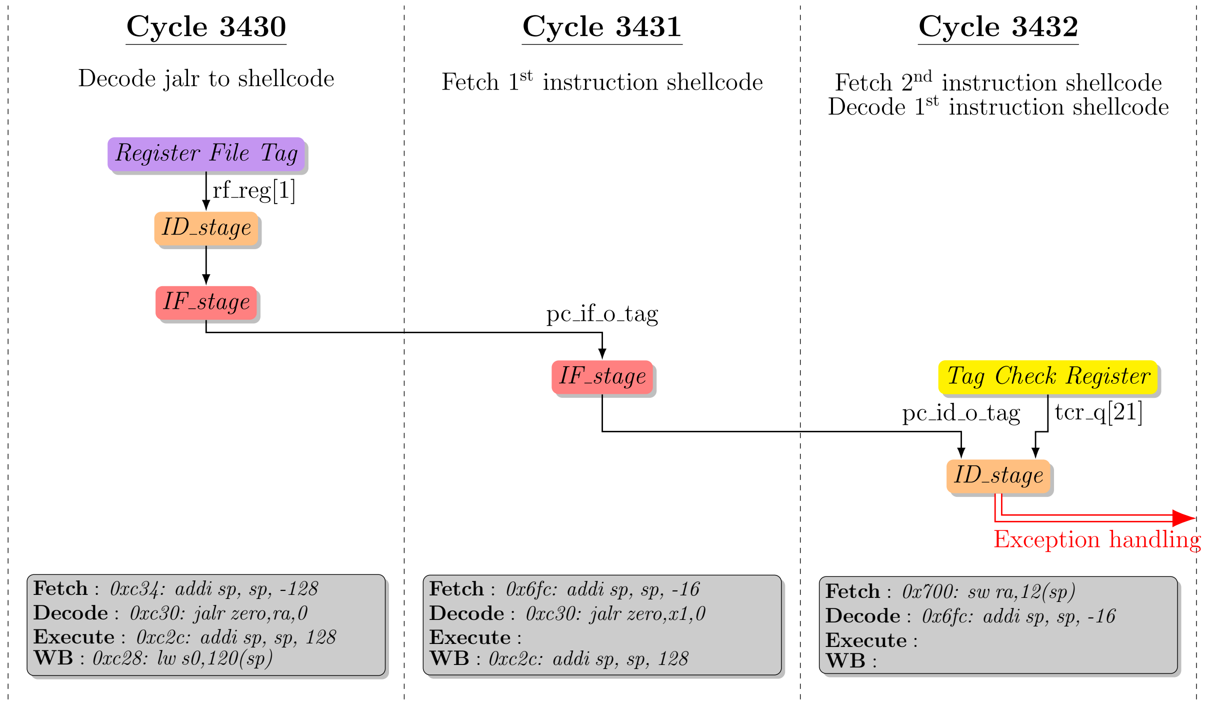
3.3.2 First use case: Buffer overflow

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 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 3.3 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 3.3 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 3.3 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.



```
Fetch : 0xc34: addi sp, sp, -128
Decode : 0xc30: jalr zero,ra,0
Execute : 0xc2c: addi sp, sp, 128
WB : 0xc28: lw s0,120(sp)
```

```
Fetch : 0x6fc: addi sp, sp, -16
Decode : 0xc30: jalr zero,x1,0
Execute :
WB : 0xc2c: addi sp, sp, 128
```

```
Fetch : 0x700: sw ra,12(sp)
Decode : 0x6fc: addi sp, sp, -16
Execute :
WB :
```

Figure 3.3: Tag propagation in a buffer overflow attack

To further study the propagation of the fault, Figure 3.4 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 3.4 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 3.4, 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 3.4 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.

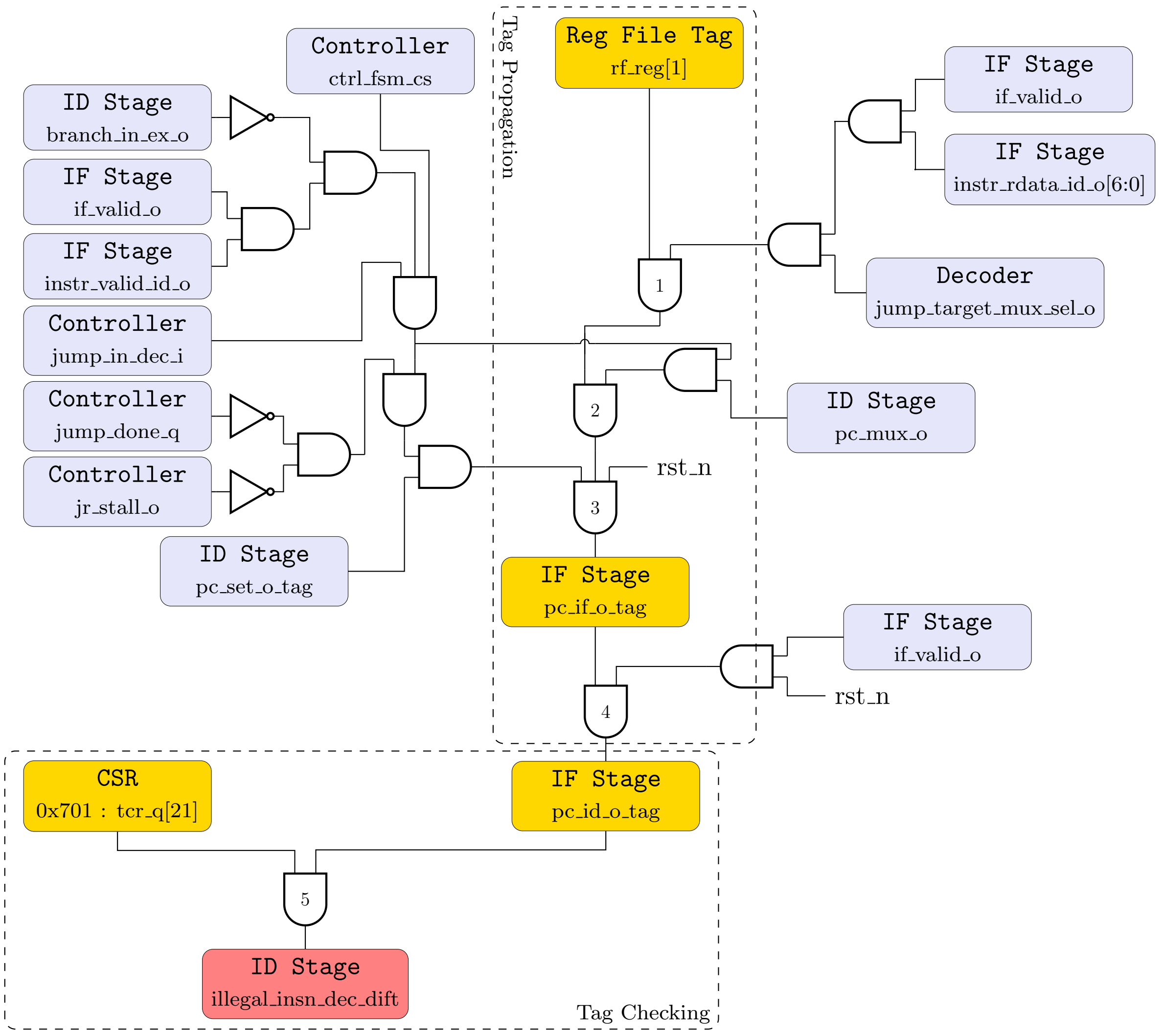


Figure 3.4: Logic description of the exception driving in a buffer overflow attack

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 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 3.5 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 3.5, drives a multiplexer leading to the propagation of register *store_dest_addr_ex_o_tag*.

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	bit-flip	set0	set1	bit-flip	set0	set1	bit-flip	set0	set1	bit-flip	set0	set1	bit-flip	set0	set1	bit-flip	set0	set1	bit-flip	set0	set1	bit-flip
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																						✓		✓

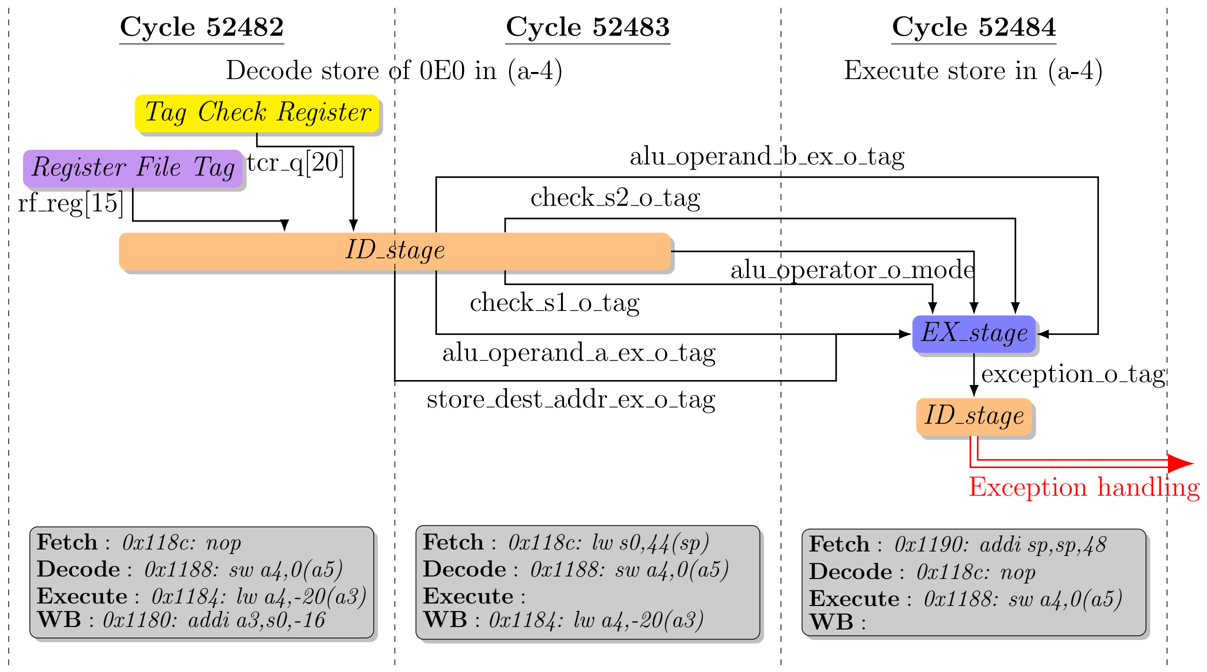


Figure 3.5: Tag propagation in a format string attack

To further study the propagation of the fault, Figure 3.6 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 3.6 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 3.6, 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 3.6 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'.

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	bit-flip	set0	set1	bit-flip	set0	set1	bit-flip	set0	set1	bit-flip
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											✓	✓

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 tagged variable in the code (variable `a`). Finally, observations for both *tpr_q* and *tcr_q* are similar than 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 3.7 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 behaviour is driven by *check_s1_o_tag* through *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 3.7 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 3.7. This register drives a multiplexer leading to tag propagation presented in Figure 3.7.

To further study the faults' propagation, Figure 3.8 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). Figure 3.8 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 3.8, 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 3.8 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 chapter, we described the processor we focus on with its implementation of a hardware in-core DIFT. We described how it works and how to use the DIFT mechanism with the default configuration. Then, we described the different use cases we choose to work with, in order to analyse the DIFT behaviour and assess it against fault injection attacks. Finally, we presented the vulnerability assessment on these use cases using the D-RI5CY security mechanism. We shown that this DIFT implementation is vulnerable to FIA within different registers depending on the fault model and depending on the application, as different paths are used and so different registers are going to be criticals.

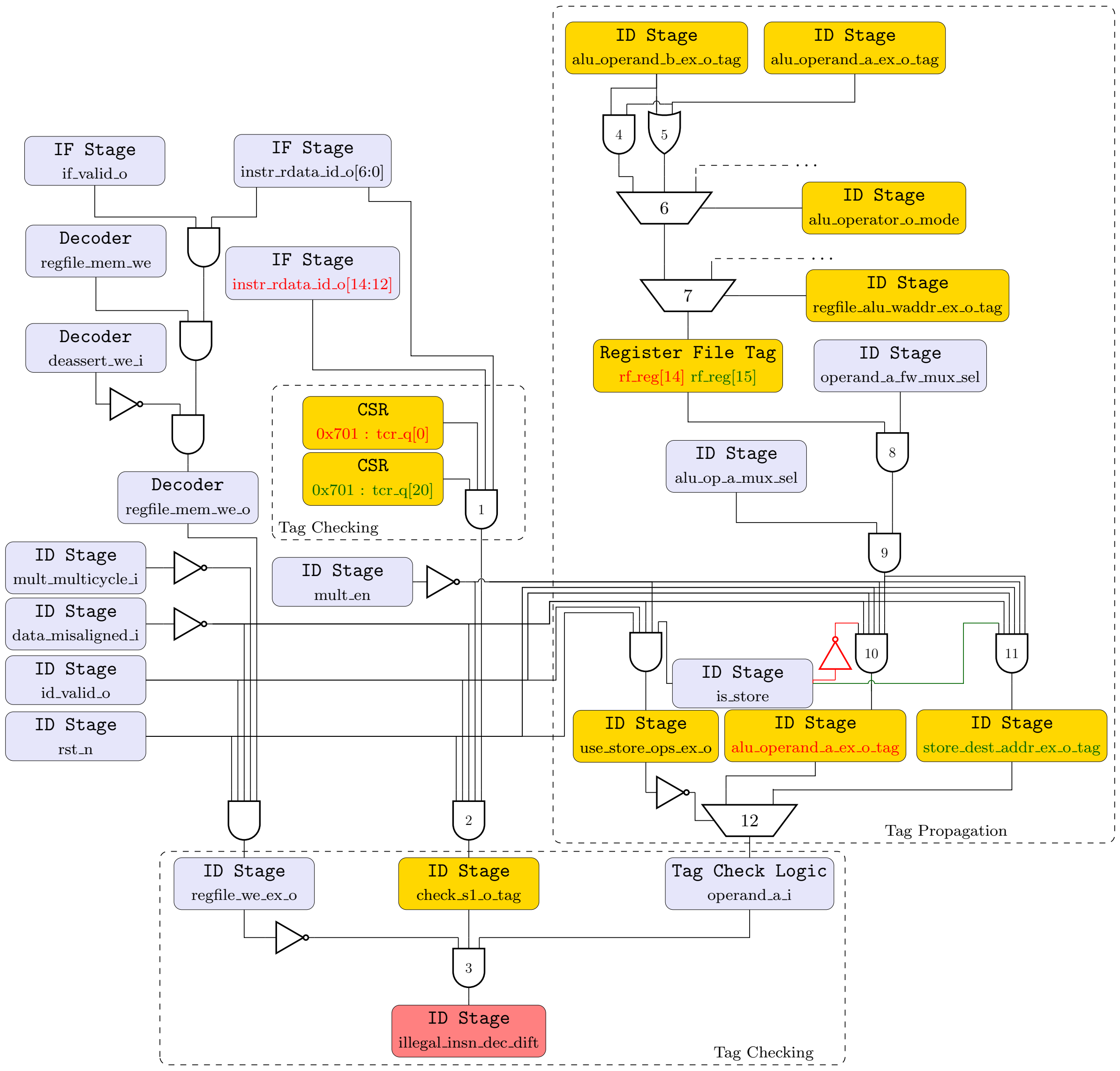


Figure 3.6: Logic description of the exception driving in a format string attack

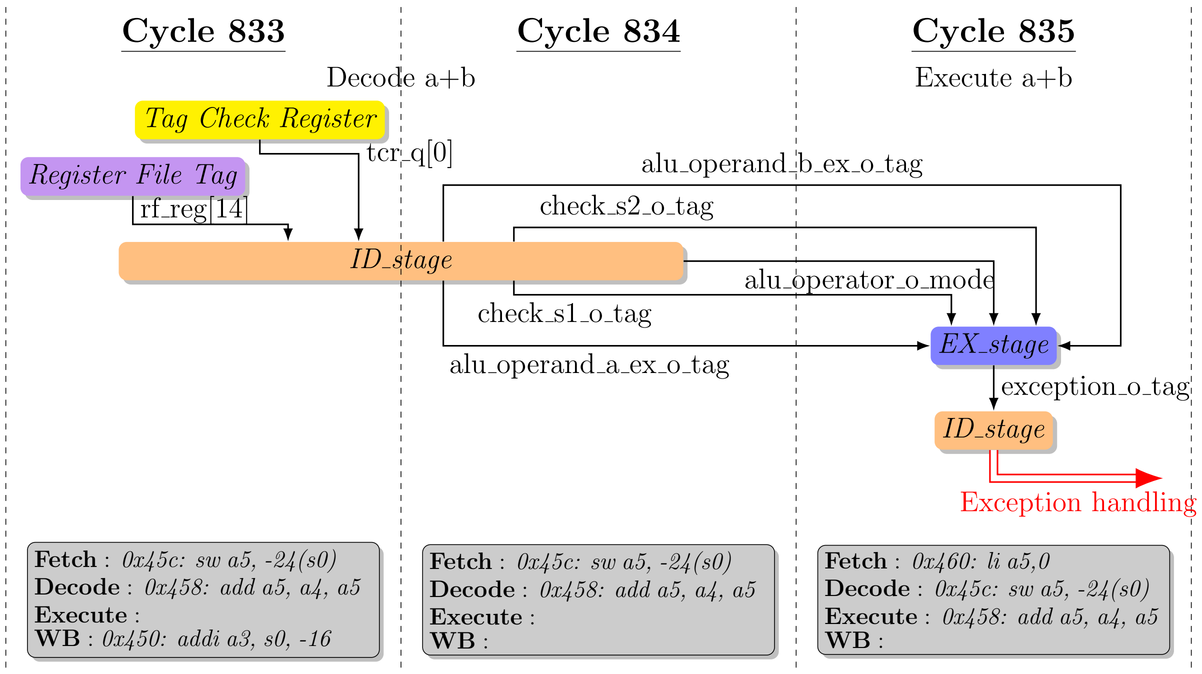


Figure 3.7: Tag propagation in a computation case with the compare/compute use case

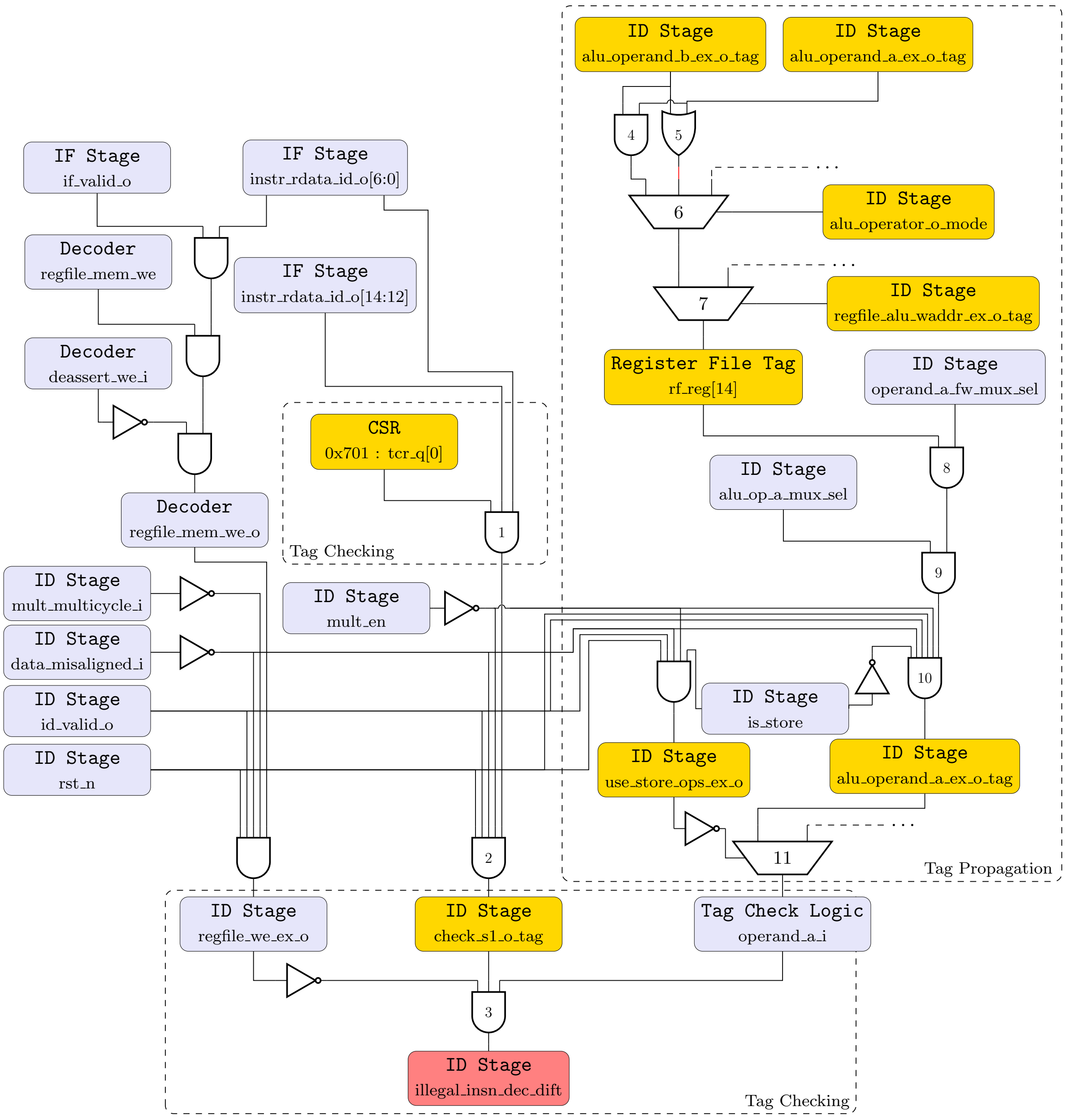


Figure 3.8: Logic representation of tag propagation in a computation case

Chapter 4

FISSA – Fault Injection Simulation for Security Assessment

Contents

This chapter introduces and presents a tool, called FISSA – Fault Injection Simulation for Security Assessment –, created to automate fault injection attacks campaigns in simulation. The first section presents the state of the art of existing tools for FIA campaigns in emulation, formal methods or even perform real world attacks. The second section presents the architecture and details how FISSA works and presents how to extend it depending on other needs. The third section presents an example to present how FISSA work in real conditions with a use case from Section 3.2. Finally, we will discuss and draw some perspectives for the tool's development and usability.

4.1 Simulation tools for Fault Injection

Addressing fault injection vulnerabilities is crucial. Historically, fault attacks have been conducted using physical equipment. Nonetheless, a modern approach has emerged that leverages simulators for fault testing. The main advantages of using simulators are they cost less money than physical setups, it is easier to make them work as they do not need specific skills, and they can be used during the conceptual stage.

Table 4.1: Fault Injection based methods for vulnerability assessment comparison

	References	Cost	Control over fault scenarios	Scalability	Speed of execution	Realism	Expertise
Formal Methods	[33–36]	Very low	Very high	Very low	Low	Low	Very high
Simulations	[37–39]	Very low	Very high	Low	Low/Moderate	Moderate	Low
Emulations	[40–43]	High	Moderate	High	Very high	High	Moderate
Actual FIA	[29, 44–46]	Very high	Very low	Very high	Very high	Very high	Very high

This section presents recent works related to methods and tools for vulnerability assessment when considering fault injection attacks. For such vulnerability assessment, main strategies include actual fault injections, emulations, formal methods and simulations. **William** ►*Ajouter état de l'art plus complet sur cette partie*◄

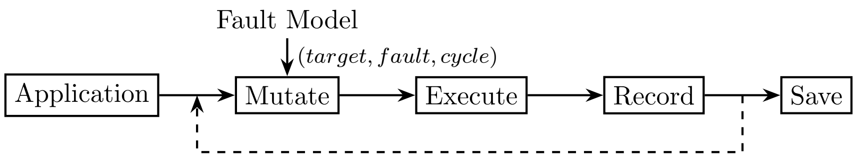


Figure 4.1: Anatomy of a Fault Injection tool

The diagram presented in Figure 4.1 illustrates the process of a fault injection. The process begins with an *Application*, which represents the target system under test. This application is then passed to the *Mutate* state, where faults are introduced based on a predefined *Fault Model*. This fault model provides parameters such as the `target` to be attacked or the list of targets, the type of `fault` (set to 0/1, bit-flip, etc), and the `clock cycle` that guide the mutation process.

After the fault is injected, the faulted application is executed, the *Execute* state. The results of this execution are captured by the *Record* module, which logs the outcomes for further analysis.

Finally, the recorded data is saved in the *Save* module, completing the fault injection cycle. The dashed arrow from the *Record* module back to the *Mutate* module indicates a feedback loop, where the results of the execution can influence subsequent fault injections. This iterative process helps in thoroughly evaluating the resilience and fault tolerance of the application.

Actual FIAs involve physically injecting faults into the target hardware using techniques such as variations in supply voltage or clock signal [29, 44], laser pulses [29, 46], electromagnetic emanations [29] or X-Rays [45]. This approach offers valuable insights into the real impact of faults on hardware components. However, a significant drawback of actual fault injections is that they demand considerable expertise to prepare the target, involving intricate setup procedures. Additionally, this approach can only be executed once the physical circuit is available, potentially delaying the vulnerability assessment process until later stages of development.

Fault emulation can, for instance, rely on FPGA [40], or on an emulator such as QEMU [41, 42] to perform fault injection campaigns. This approach is four times faster than simulation-based techniques [43], and unlike simulation-based or formal method-based fault injections techniques, the size of the evaluated circuit has no major impact on the fault injection campaign timing performances. However, configuring an emulation environment can be complex and time-consuming. Achieving an accurate representation of the target system may require detailed configuration and parameter tuning. The accuracy of emulation is contingent on the quality of the models used to replicate the target hardware. If the models are inaccurate or incomplete, the results of fault injections may not precisely reflect actual behaviour.

Formal methods provide an advantage with mathematical proofs, ensuring a rigorous verification of the system's behaviour during fault injection experiments. Formal methods approaches such as [33] allow the analysis of a circuit design in order to detect sensitive logic or sequential hardware elements. [34], [35] and [36] present formal verification methods to analyse the behaviour of HDL implementation. However, this type of tool usually suffers from restrictions limiting its actual usage on a complete processor. Conventional formal approaches encounter scalability challenges due to limitations in verification techniques. In particular, the circuit structure it can analyse is usually limited.

Fault Injections simulations can be performed at processor instructions level. Authors of [37] explore the impact of fault injection attacks on software security. They evaluate four open-source fault simulators, comparing their techniques and suggest enhancing them with AI methods inspired by advances in cryptographic fault simulation. [38] is an open-source deterministic fault attack simulator prototype utilising the Unicorn Framework and Capstone disassembler. [39] introduces VerFI, a gate-level granularity fault simulator for hardware implementations. For instance, it has been used to spot an implementation mistake in ParTI [47]. However, this tool has been developed to check if implemented countermeasures can really protect against fault injection on cryptographic implementations, but it cannot evaluate components such as registers or memories. In this paper, we focus on CABA simulations, which provides a controlled virtual environment for injecting faults. There are several solutions of simulations in an HDL simulator like Questasim, Vivado, etc. *Behavioural* simulation is used to detect functional issues and ensuring that the design behaves as expected. *Post-synthesis* simulation verifies that the synthesised netlist matches the expected functionality. *Timed* simulation is used to ensure that the design meets timing requirements and can operate at the specified clock frequency. And finally, *post-implementation* simulations are used to verify that the implemented design meets all requirements and constraints, including those related to the physical layout on the target. Simulation-based fault injection offers the advantage of enabling designers to test their system throughout the design cycle, providing valuable insights and uncovering potential vulnerabilities early in the development process. However, a limitation lies in the potential lack of absolute fidelity to actual conditions, as simulations might not perfectly replicate all hardware intricacies, introducing a slight risk of overlooking certain faults that could manifest in the actual hardware.

Table 4.1 shows a comparison between these four methods for vulnerability assessment when considering FIA regarding six metrics. These metrics are the financial cost of setting up the fault injection campaign, the control over fault scenarios (how configurable are the scenarios), scalability which refers to the method capacity to be applied to systems of different sizes or complexities, speed of execution of the campaign, realism of the fault injection campaign and the level of required expertise. Table 4.1 shows that no method is completely optimal. Each method has its own advantages and disadvantages and must be chosen by the designer according to the requirements and the available financial and human resources. Indeed, setting up an actual fault injection campaign requires much more expertise in this domain and also requires costly equipment, whereas setting up a simulation campaign can be easier for a circuit designer familiar with HDL simulation tools such as Questasim. Table 4.1 shows that CABA simulation offers a good compromise to assess the security level of a circuit design. In particular, it provides an efficient solution for investigating security throughout the design cycle, enabling the concept of "Security by Design".

4.2 FISSA

This section presents our open-source tool, FISSA, available on GitHub [48] under the CeCILL-B licence.

4.2.1 Main software architecture

FISSA is designed to help circuit designers to analyse, throughout the design cycle, the sensitivity to FIA of the developed circuit. Figure 4.2 presents the software architecture of FISSA. It consists of three different modules: *TCL generator*, *Fault Injection Simulator* and *Analyser*. The first and third modules correspond to a set of Python classes.

The TCL generator, detailed in Section 4.2.3, relies on a configuration file and a target file to create a set of parameterised TCL scripts. These scripts are tailored based on the provided configuration file and are used to drive the fault injection simulation campaign.

Fault Injection Simulator, detailed in Section 4.2.4, performs the fault injection simulation campaign based on inputs files from *TCL generator* for a circuit design described through HDL files and memory initialisation files. For that purpose it relies on an existing HDL simulator such as Questasim [49], Verilator [50], or Vivado [51] to simulate the design according to the TCL script and generates JSON files to log each simulation.

The Analyser, detailed in Section 4.2.5, evaluates the outcomes of the simulations and generates a set of files that allows the designers to examine fault injection effects on their designs through various information.

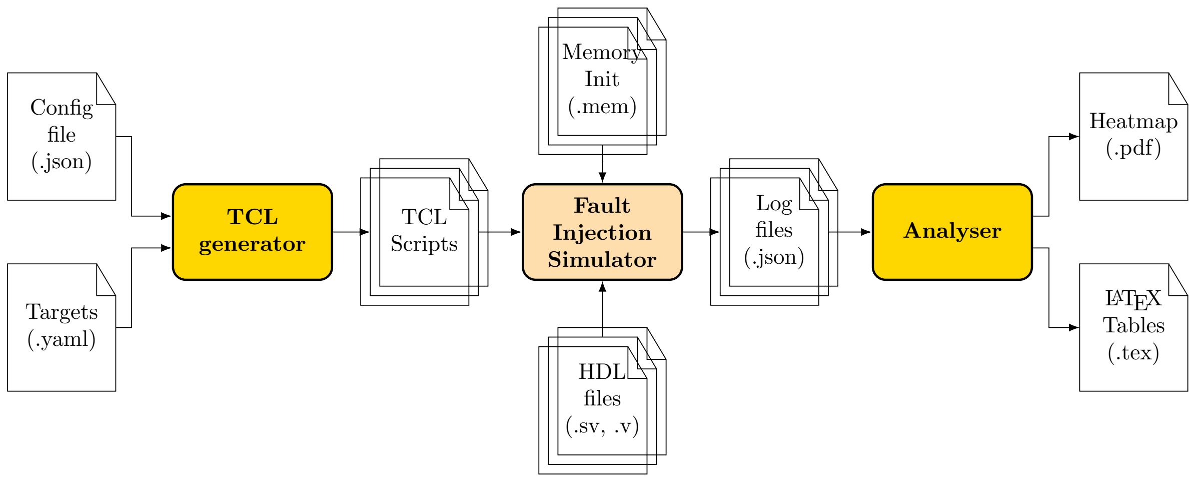


Figure 4.2: Software architecture of FISSA

Algorithm 1 shows a representation of a fault injection campaign. The algorithm requires a set of targets (i.e. hardware elements in which a fault should be injected), the fault model and the considered injection window(s) which identifies the period(s), in number of clock cycles, in which fault injections are performed. Then, it runs a first simulation with no fault injected, which is used as a reference for comparison with the following simulations to determine end-of-simulation statuses. Then, for each target, each fault model and for each clock cycle within the injection window, the corresponding simulation is executed, and the corresponding logs are stored in a dedicated file.

Customising end-of-simulation statuses allows for adaptation to the specific requirements of each design assessment. To configure these statuses, adjustments need to be made either directly in FISSA's code or the HDL code. This process may involve evaluating factors such as:

- hardware element content (signals, registers, ...),
- simulation time (e.g. the simulation exceeds a reference number of clock cycles),
- simulation's end (e.g. an assert statement introduced in the HDL code is reached)

Algorithm 1 Simulated FIA campaign pseudo-code

```
Require: $targets \leftarrow list(targets)$
Require: $faults \leftarrow list(fault\_model)$
Require: $windows \leftarrow list(injection\_windows)$
1: $ref\_sims = simulate()$
2: for $target \in targets$ do
3:     for $fault \in faults$ do
4:         for $cycle \in windows$ do
5:             $logs = simulate(target, fault, cycle)$
6:         end for
7:     end for
8: end for
```

4.2.2 Supported fault models

A set of fault models has already been integrated into FISSA for different needs. For a given fault injection campaign, the relevant fault model is defined in the input configuration file and is applied to targets during the simulation phase. Currently, supported fault models are:

- target set to 0/1: for each cycle of the injection window and for each target, we set them individually to 0 or 1, in turn exhaustively ($nbSimulations = nbCycles * nbTargets$),
- single bit-flip in one target at a given clock cycle: for each cycle of the injection window, we do a bit-flip for each bit of every targets exhaustively ($nbSimulations = nbCycles * nbBits$),
- single bit-flip in two targets at a given clock cycle: we take one cycle and a couple of targets' bits (it can be the same target at two different bits) and we bit-flip these two bits ($nbSimulations = nbCycles * C_2^k$; with k, the total number of bits in the attacked system),
- single bit-flip in two targets at two different clock cycles: we take two different cycles and a couple of targets' bits (it can be the same target at two different bits) and we bit-flip these two bits ($nbSimulations = C_2^{nbCycles} * C_2^k$; with k, the total number of bits in the attacked system),

- exhaustive multi-bits faults in one target at a given clock cycle: we take one cycle and one target and we try exhaustively each combinations of bits (for example for a 2 bits target, it would be: 00, 01, 10, 11) and we set the target at each value ($nbSimulations = nbCycles * 2^{targetSize1}$). It is worth nothing that for this fault model, we only take targets between 1 and 16 bits to avoid very big numbers of simulations as 2^{32} would be too long to simulate exhaustively,
- exhaustive multi-bits faults in two targets at a given clock cycle: we take one cycle and two targets and we try exhaustively each combinations of bits (for example for a 2 bits target, it would be: 00, 01, 10, 11) for each target and we set them to each value ($nbSimulations = nbCycles * 2^{targetSize1} * 2^{targetSize2}$). It is worth nothing that for this fault model, we only take targets between 1 and 10 bits to avoid very big numbers of simulations as 2^{32} would be too long to simulate exhaustively.

4.2.3 TCL Generator

Listing 4.1: Example of a FISSA configuration file

```
{
    "name_simulator": "modelsim",
    "path_tcl_generation": "PATH/",
    "path_files_sim": "PATH/simu_files/",
    "path_generated_sim": "PATH/simu_files/generated_simulations/",
    "path_results_sim": "PATH/simu_files/results_simulations/",
    "path_simulation": [ "PATH_SIMU/"],
    "prot": "wop",
    "version": 1,
    "name_reg_file_ext_wo_protect": "/faulted-reg.yaml",
    "application": ["buffer_overflow", "secretFunction", "propagationTagV2"],
    "name_results": {
        "buffer_overflow": "Buffer Overflow",
        "secretFunction": "WU-FTPd",
        "propagationTagV2": "Compare/Compute"
    },
    "threat_model": [
        "single_bitflip_spatial"
    ],
    "multi_fault_injection": 2,
    "avoid_register": [],
    "avoid_log_registers": [],
    "log_registers": [],
    "injection_window": {
        "buffer_overflow": [
            [137140, 137380]
        ],
        "secretFunction": [
            [2099100, 2099420]
        ],
        "propagationTagV2": [
            [33300, 33460]
        ]
    },
    "cycle_ref": 100,
    "cpu_period": 40,
    "batch_sim": {
        "buffer_overflow": 2000,
        "secretFunction": 2000,
        "propagationTagV2": 2000
    },
    "multi_res_files": {
        "buffer_overflow": 8,
        "secretFunction": 8,
        "propagationTagV2": 8
    }
}
```

The *TCL Generator* is used to generate the set of TCL script files which drive the *fault injection simulator*. This module requires two input files. Figure 4.3 details the *TCL Generator*. Each blue box represents a python class used to generate the set of output TCL scripts. The *initialisation* class gets inputs from a configuration file. This JSON-formatted file includes various parameters such as the targeted HDL simulator, the considered fault model and the injection window(s). Furthermore, it encompasses parameters such as the clock period (in ns) of the HDL design and the maximum number of simulated

Listing 4.2: Example of a FISSA target file

```yaml
---
## FETCH
FETCH:
    -
        name: /tb/top_i/core_region_i/RISCV_CORE/if_stage_i/pc_id_o_tag
        width: 1
    -
        name: /tb/top_i/core_region_i/RISCV_CORE/if_stage_i/pc_if_o_tag
        width: 1

## DECODE
DECODE:

## RF TAG
RF_TAG:

## EXECUTE
EXECUTE:

## CSR
CSR:

## Load Store Unit
LSU:
 ...
```

clock cycles used to stop the simulation in case of divergence due to the injected fault. Moreover, one extra parameter defines the quantity of simulations per TCL file, allowing a simulation parallelism degree. Listing 4.1 shows an extract of a configuration file used for our fault injection campaigns. Listing 4.2 shows an extract from a target file according to the configuration file provided previously. This file list each stage of the RISC-V core and for each the HDL path of our targets are writen. Here, in this example, only the list of targets for the *instruction fetch* stage is listed.

The *Targets* file contains, in YAML format, the list of the circuit elements (e.g. registers or logic gates) that need to be targeted during the fault injection campaign. For each target, its HDL path and bit-width are specified. *TCL Script Generator* class gets the configuration parameters from *Initialisation* class, reads the *Targets'* file and calls three others classes. The first one, *Basic Code Generator*, undertakes the fundamental generation of TCL code for initialising a simulation, running a simulation, and ending a simulation. The second one, *Fault Generator*, produces the TCL code related to fault injection. The *TCL Script Generator* provides specific parameters to the *Fault Generator* to produce code for a designated set of targets and a specified set of clock cycles for fault injection. The third one, *Log Generator*, produces the TCL code to produce logs after each simulation. Logs comprise the simulation's ID, fault model, faulted targets, injection clock cycle(s), end-of-simulation status, values for all targets, and the end-of-simulation clock cycle. This data constitutes the automated aspect of logging. Finally, the *TCL Script Generator* outputs a set of TCL files, each one correspond to a batch of simulations. This allows the user to perform a per batch results analysis. It is worth noting that each batch starts with a reference simulation which means a simulation without any fault injected. It allows to have results for comparison after when a fault occured and determine what happened due to the injected fault. Additionally, it generates a target file utilised by TCL scripts to obtain a simplified target list (refer to Subsection 4.2.4), as the simulation log requires a list of targets without their sizes.

William ▶*Modification des Listings 4.1, 4.2, 4.3 en les remplaçant par un exemple simple genre additionneur (à la place de le mettre dans la section 4.3)?*◀

Algorithm 2 depicts a fault injection simulation pseudo-code, showcasing requirements, and each state with essential parameters. Additionally, the corresponding Python class from Figure 4.3 is added for each line. Line 5 in Algorithm 1 corresponds to Algorithm 2. This algorithm is executed multiple times with

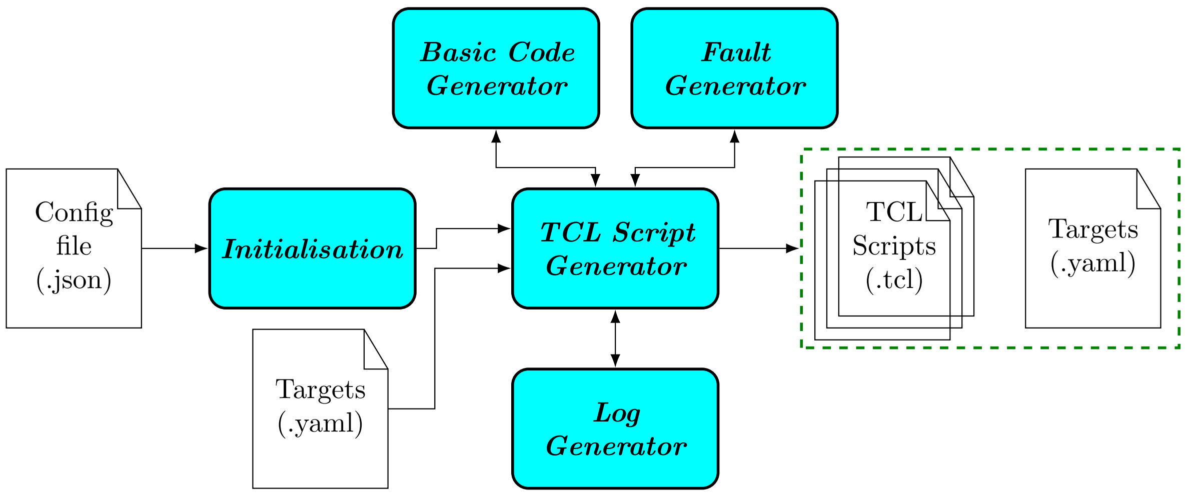


Figure 4.3: Software architecture of the TCL Generator module

different inputs to build a TCL script.

Algorithm 2 FIA simulation pseudo-code

```
Require: $target$
Require: $cycle$
Require: $fault\_model$
1: $tcl\_script = init\_sim(fault\_model, cycle, target)$ // generated by Basic Code Generator
2: $tcl\_script += inject\_fault(fault\_model)$ // generated by Fault Generator
3: $tcl\_script += run\_sim()$ // generated by Basic Code Generator
4: $tcl\_script += log\_sim(fault\_model)$ // generated by Log Generator
5: $tcl\_script += end\_sim()$ // generated by Basic Code Generator
6: $tcl\_file.write(tcl\_script)$ // append and write the simulation data inside the TCL file
```

4.2.4 Fault Injection Simulator

The *Fault Injection Simulator* mainly relies on an existing HDL simulator to perform simulations by executing the TCL scripts produced by the *TCL generator*. The log files, in JSON format, are generated by the TCL script for each simulation. This file encompasses data such as the current simulation number, the executed clock cycle count, the values of the targets' file, the targets faulted, the fault model and the end-of-simulation status.

Listing 4.3 shows a simplified example of an output file from a simulation. Many lines are omitted to simplify the text and its comprehension. In this example, we have the result of the first simulation of the campaign. The fault model is a single bit-flip in one target at a given clock cycle, and the target, which is a register in this case, `pc_id_o_tag`, has a size of one bit. We attack it at the period time of 137,140 ns. The omitted lines, at line 7, include all registers from the register file, all register file tags, and all registers from the target list. The last line, line 14, shows that this simulation ended with a status equal to 3 (i.e., exception delayed from the reference simulation).

It is worth noting that the set of calls to the generated TCL scripts has to be integrated into the designer's existing design flow, allowing the design compilation, initialisation, and management of input stimuli. The use of TCL scripts simplifies such an integration. Once all the fault injection simulations have been performed, the log files can be sent to the *Analyser* which, is described in the following subsection.

Listing 4.3: Extract of an example of a FISSA output log JSON file

```
"simulation_1": {
    "cycle_ref": 100,
    "cycle_ending": 4,
    "TPR": "32'h0000a8a2",
    "TCR": "32'h00341800",
    "rf1": "32'h000006fc",
    (...)
    "faulted_register": "/tb/top_i/core_region_i/RISCV_CORE/if_stage_i/pc_id_o_tag",
    "size_faulted_register": 1,
    "threat": "bitflip",
    "bit_flipped": 0,
    "cycle_attacked": "137140 ns",
    "simulation_end_time": "137300 ns",
    "status_end": 3
}
```

4.2.5 Analyser

The *Analyser* reads all log files and generates a set of LaTeX tables (*.tex* files) and/or sensitivity heatmaps (in PDF format) according to the fault models, allowing the user to identify the sensitive hardware elements in the circuit design. The generated tables can be customised through modification in the *Analyser* Python code. The current configuration captures and counts the diverse end-of-simulation status. Heatmaps are generated for multi-target fault models. For instance, when considering a 2 faults scenario disturbing two hardware elements, a 2-dimension heatmap allows the user to identify sensitive couples of hardware elements leading to a potential vulnerability. Their configuration can be adapted by modifying the *Analyser* Python code. Heatmaps generation is based on *Seaborn* [52] which relies on *Matplotlib* [53]. This library provides a high-level interface for drawing attractive and informative statistical graphics and save them in different formats like PDF, PNG, etc. In the current configuration, heatmaps highlight the targets leading to a specific end-of-simulation status (e.g. a status identified by the designer as a successful attack). Once the results have been generated, they can easily be inserted into a vulnerability assessment report.

4.2.6 Extending FISSA

In order to extend FISSA for integrating an additional fault model, some modifications to the *TCL Script Generator*, the *Basic Code Generator*, the *Fault Generator* and *Log Generator* modules are necessary. It requires the extension of the *init_sim*, *inject_fault* and *log_sim* functions presented in Algorithm 2 to implement the new fault model from initialisation to logging. For instance, these extensions should define the targets for each simulation, the impact of the injections (set to 0/1, bit-flip, random, etc) and the set of data to be logged for this fault model. The *Log Generator* automates the extraction of specific segments from the ongoing simulation. However, it is customisable, enabling the modification of logged elements, such as incorporating memory content or a list of signals.

Analyser can be extended to produce additional LaTeX tables, heatmaps or any other way of results visualisation. This can be achieved by either modifying the existing methods or by developing new ones.

An integral aspect of expanding FISSA involves adjusting functions depending on the used HDL simulator. Despite the definition of the TCL language, specific commands vary between simulators.

4.3 Use case example

This section presents a case study to demonstrate the use of FISSA in real conditions. It focuses on the evaluation of the robustness of the DIFT mechanism integrated in the D-RI5CY processor with the Buffer overflow use case from Section 3.2.

William ►*Est ce que je laisse cet exemple qui est le même que celui de DSD ou j'ajoute un exemple plus simple où j'attaque un additionneur avec 3 registres avec quelques modèles de fautes ?*◄

4.3.1 FISSA's configuration

This subsection presents FISSA's configuration for the addressed use case. We have defined four end-of-simulation statuses, which will be utilised to automatically generate results tables. Examples of these tables will be provided in Subsection 4.3.2. The initial status is labelled as a *crash* (status 1), indicating that the fault injection has caused a deviation in program flow control, leading the processor to execute instructions different from those expected. The second status, identified as a *silent* fault (status 2), signifies that a fault has occurred but has not impacted the ongoing simulation behaviour. Status 3, termed a *delay*, denotes that the fault has delayed the DIFT-related exception, meaning the exception is not raised at the same clock cycle as in the reference simulation. The final status is referred to as a *success* (status 4), indicating a bypass of the DIFT mechanism and thereby marking a successful attack. This status corresponds to the detection of the end of the simulated program, with no exception being raised.

In the input configuration file, a single injection window is set between cycles 3428 and 3434, the maximum number of simulated clock cycles is set to 100 from the start of the injection window, this allows us to detect if there were a control flow deviation, the design period is set to 40 ns, the number of simulations per TCL script is set to 2,200. The considered fault models are the seven fault models defined in Section 4.2.2: *target set to 0*, *target set to 1*, *single bit-flip in one target at a given cycle*, *single bit-flip in two targets at a given cycle*, *single bit-flip in two targets at two different cycles*, *exhaustive multi-bits faults in one target at a given cycle*, *exhaustive multi-bits faults in two targets at a given cycle.*

Seven FIA simulation campaigns are performed to evaluate the design against the seven fault models. We choose to log the values of the *Targets'* file, the simulation's number, targets' value after the injection, the injection cycle and the end-of-simulation status. The *Targets'* file is filled with the 55 registers of the DIFT security mechanism representing a total of 127 bits in total.

4.3.2 Experimental results

This section presents results obtained using FISSA on the considered use case. All experiments are performed on a server with the following configuration: Xeon Gold 5220 (2,2 GHz, 18C/36T), 128 GB RAM, Ubuntu 20.04.6 LTS and Questasim 10.6e.

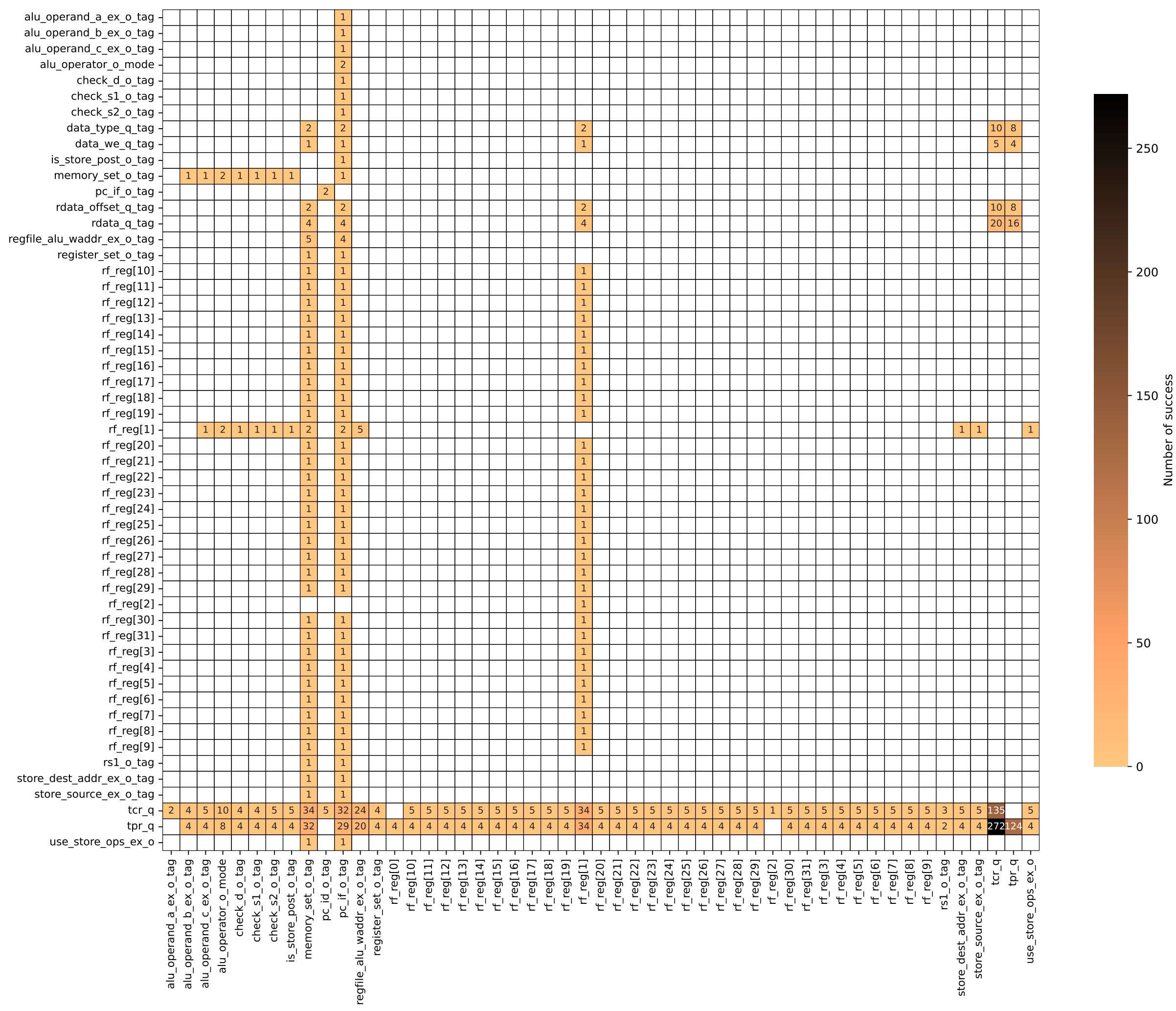


Figure 4.4: Heatmap generated according to the single bit-flip in two targets at a given clock cycle fault model

Table 4.2 summarises the outcomes of the seven previously described fault injection campaigns, with each row representing a distinct fault model. Table 4.2's columns delineate the potential end statuses for each simulation. This table is an essential tool for the designers, enabling them to analyse the vulnerabilities associated with each fault model within their design. Consequently, the designers can determine the necessity for additional protective measures or design alterations.

For instance, Table 4.2 illustrates that the '`set to 1`' fault model results in only three successful outcomes, whereas the '`single bit-flip in two targets at two different clock cycles`' fault model leads to 2,159 successes. These findings guide the designers in evaluating the significance of protecting against specific fault models.

To further assess vulnerabilities, the designers can utilise Table 4.3, which provides detailed information on the register and cycle locations of faults for models with fewer successful outcomes. For fault

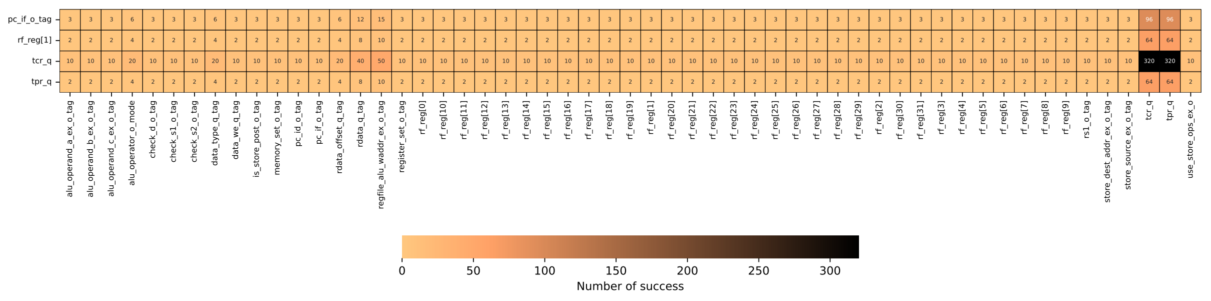


Figure 4.5: Heatmap generated according to the single bit-flip in two targets at two different clock cycles fault model

Table 4.2: Results of fault injection simulation campaigns

Fault model	Crash	Silent	Delay	Success	Total
Set to 0	0	320	1	9 (2.73%)	330
Set to 1	0	320	7	3 (0.91%)	330
Single bit-flip in one target at a given clock cycle	0	738	12	12 (1.57%)	762
Single bit-flip in two targets at a given clock cycle	0	45,097	1,503	1,406 (2.93%)	48,006
Single bit-flip in two targets at two different clock cycles	0	238,633	1,143	2,159 (0.89%)	241,935
Exhaustive multi-bits faults in one target at a given clock cycle	0	927	6	3 (0.32%)	936
Exhaustive multi-bits faults in two targets at a given clock cycle	0	67,072	926	450 (0.66%)	68,448

models with a high number of successes, where the table may become unwieldy, Figure 4.4 serves as a more accessible reference. This figure helps in visualising and interpreting the spatial distribution of vulnerabilities effectively.

Table 4.3 is produced by FISSA and details the successes from three distinct fault injection campaigns: `set to 0`, `set to 1` and `single bit-flip in one target at a given cycle`. Table 4.3 specifies successes for each fault model, correlated with the cycle and the affected target. For example, a `set to 0` fault at cycle 3428 on `tcr_q` would lead to a successfully attack. It highlights which targets are sensitive to fault attacks at a cycle-accurate and bit-accurate level, providing the designers precise information on critical elements requiring protection based on their specific needs. Table 4.3 only covers the most basic fault models. Indeed, producing a table for more complex scenarios, such as simultaneous faults in two targets within a same or multiple cycles, would be intricate and challenging to interpret. Consequently, we opted for an alternative method and developed a heatmap representation (e.g. Figure 4.4).

To further explore the impact of FIA on a design, a designer can study heatmaps generated by FISSA. These heatmaps are tailored to a fault model with two faulty registers, where each matrix intersection shows the number of successes with that target pair.

Figure 4.4 shows the heatmap generated for the single bit-flip in two targets at a given clock cycle fault model. The colour scale represents the number of fault injections targeting a couple of hardware elements (i.e. registers for this use case) leading to a *success* as defined in Subsection 4.3.1. We can note that this colour scale, in our case, range from 1 to 272 with 0 excluded. This figure highlights the registers that are critical to a specific fault model, allowing the designer to assess his design and choose which protection and where a protection is required, from low need to very high need. To give an example, it

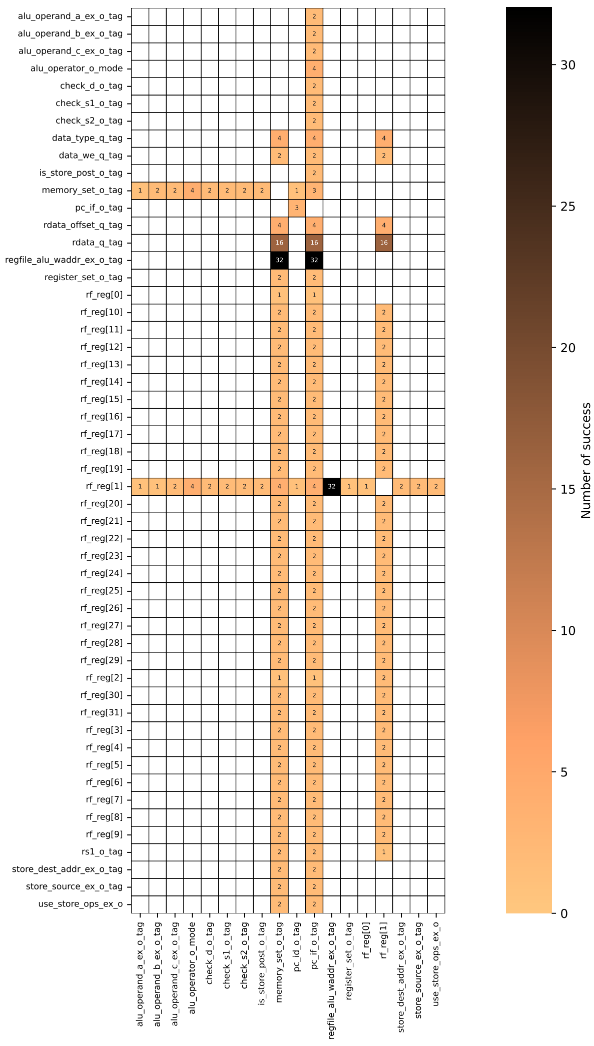


Figure 4.6: Heatmap generated according to the exhaustive multi-bits faults in two targets at a given clock cycle fault model

can be noted that the horizontally displayed registers `tcr_q` and `tpr_q` are critical registers, because a success will occur regardless of the associated register. Similarly, the registers shown vertically, `memory_set_o_tag`, `pc_if_o_tag`, and `rf_reg[1]`, are also critical because they lead to many successes with almost all tested registers.

To provide an analytical perspective from the buffer overflow use case presented in Section 3.2, the five previously mentioned registers are critical as they either store the DIFT security policy configuration (`tpr_q` and `tcr_q`) or store (`rf_reg[1]` represents the tag associated with the value of the Program Counter (PC), which is stored in the register file at index 1 for RISC-V ISA) and propagate the tag (`pc_if_o_tag`) associated with the PC. This is particularly important in our example, which demonstrates an ROP attack via a buffer overflow. The colour scale indicates the impact of the fault injections on the combination of registers tested. For example, a pair associated with a high number such as 272, 124, and 135

Table 4.3: Buffer overflow: success per register, fault type and simulation time

	Cycle 3428			Cycle 3429			Cycle 3430			Cycle 3431			Cycle 3432		
	set 0	set 1	bit-flip	set 0	set 1	bit-flip	set 0	set 1	bit-flip	set 0	set 1	bit-flip	set 0	set 1	bit-flip
pc_if_o_tag										✓		✓			
memory_set_o_tag		✓	✓												
rf_reg[1]							✓		✓						
tcr_q	✓			✓			✓			✓			✓		
tcr_q[21]			✓			✓			✓			✓			✓
tpr_q	✓	✓		✓	✓										
tpr_q[12]			✓			✓									
tpr_q[15]			✓			✓									

for `tcr_q` and `tpr_q` are very high priority as they lead to 37.77% success on this fault model. In addition, we can see that several registers produce a low number of successes, such as `alu_operand_a_ex_o_tag` and `rf_reg[2]`; these registers are then not the highest priority for protection for the designer.

It allows the designer to identify the critical hardware elements to be protected for the use case under consideration. All of this information allows the designer to prioritize countermeasures according to allocated budget, protection requirements, etc.

While Table 4.2 provides the total number of *successes* for each fault model and Table 4.3 gives the successes for each fault model (*set to 0*, *set to 1*, and *a single bit flip in a target at a given cycle*) correlated with the cycle and affected target, Figure 4.4 shows that fault injections in 246 register pairs result in a *success*. This information allows the designer to focus on specific simulation traces to understand the effect(s) of the fault(s) and improve the robustness of his design by implementing adapted countermeasures.

4.4 Discussion and Perspectives

In this section, we will discuss this proposed tool and draw some perspectives for the long-term development. In terms of execution time, we did in total around 24,000,000 simulations for approximatively 3 seconds for each simulation in average spanning from initialisation to data recording. The execution time is contingent upon various parameters, including the design's size, the specific simulation case, and the number of targets involve. For example, as we have three different use cases, it goes from an average of 0.4 second to 5.8 seconds per simulation. In emulation campaigns, FPGA-based fault emulation is four times faster than simulation-based techniques, as noted in paper [43]. Actual FIAs are faster than simulations, taking about 0.35 seconds per injection in our tests, relying on the ChipWhisperer-lite platform for clock glitching injection. While simulations may be slower, they offer the benefit of not requiring an FPGA prototype or the final circuit. Furthermore, it allows integrating vulnerability assessment in the first stages of the development flow and provides a rich set of information for the designer in order to understand sources of vulnerabilities in his design.

As perspectives, we plan to extend FISSA to support new fault models and HDL simulators such as Vivado or Verilator. Additionally, we intend to enhance integration into the design workflow by adding more automatisation. This may include the management of HDL sources compilation, design's input stimuli or the development of a graphical user interface to improve the overall user experience.

4.5 Summary

In this chapter, we presented FISSA (Fault Injection Simulation for Security Assessment), our advanced and versatile open-source tool designed to automate fault injection campaigns. FISSA is engineered to seamlessly integrate with renowned HDL simulators, such as Questasim. It facilitates the execution of simulations by generating TCL scripts and produces comprehensive JSON log files for subsequent security analysis.

FISSA empowers designers to evaluate their designs during the conceptual phase by allowing them to select specific assessment parameters, including the fault model and target components, tailored to their unique requirements. The insights gained from the results generated by this tool enable designers to enhance the security of their designs, thus adhering to the principles of *Security by Design*.

Chapter 5

Countermeasures Implementations

Contents

5.1 Countermeasure 1: Simple Parity

5.2 Countermeasure 2: Hamming Code

5.3 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] Andy Greenberg, *A $500 Open Source Tool Lets Anyone Hack Computer Chips With Lasers*, URL: `https://www.wired.com/story/rayv-lite-laser-chip-hacking-tool/`.

[2] Janne Taponen, *Laser Fault Injection for The Masses*, URL: `https://blog.fraktal.fi/laser-fault-injection-for-the-masses-1860afde5a26`.

[3] David E Bell, Leonard J La Padula, et al., "Secure computer system: Unified exposition and multics interpretation", *in*: (1976).

[4] Dorothy E. Denning, "A lattice model of secure information flow", *in*: *Commun. ACM* 19.*5* (May 1976), pp. 236–243, ISSN: 0001-0782, DOI: `10.1145/360051.360056`.

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

[6] Kejun Chen et al., "Dynamic Information Flow Tracking: Taxonomy, Challenges, and Opportunities", *in*: *Micromachines* 12.*8* (2021), ISSN: 2072-666X, DOI: `10.3390/mi12080898`.

[7] G. Edward Suh et al., "Secure Program Execution via Dynamic Information Flow Tracking", *in*: *SIGPLAN Not.* 39.*11* (2004), pp. 85–96, ISSN: 0362-1340, DOI: `10.1145/1037187.1024404`.

[8] Christopher Brant et al., "Challenges and Opportunities for Practical and Effective Dynamic Information Flow Tracking", *in*: *ACM Computing Surveys* 55.*1* (Nov. 2021), ISSN: 0360-0300, DOI: `10.1145/3483790`.

[9] Ebrary, *Overview of Embedded Application Development for Intel Architecture*, URL: `https://ebrary.net/22038/computer_science/overview_embedded_application_development_intel_architecture#734`.

[10] Andrew C. Myers, "JFlow: practical mostly-static information flow control", *in*: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 228–241, ISBN: 1581130953, DOI: `10.1145/292540.292561`.

[11] Andrey Chudnov and David A. Naumann, "Inlined Information Flow Monitoring for JavaScript", *in*: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 629–643, ISBN: 9781450338325, DOI: `10.1145/2810103.2813684`.

[12] Thomas H. Austin and Cormac Flanagan, "Efficient purely-dynamic information flow analysis", *in*: *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, Dublin, Ireland: Association for Computing Machinery, 2009, pp. 113–124, ISBN: 9781605586458, DOI: `10.1145/1554339.1554353`.

[13] Vasileios P. Kemerlis et al., “libdft: practical dynamic data flow tracking for commodity systems”, *in*: *SIGPLAN Not.* 47.*7* (Mar. 2012), pp. 121–132, ISSN: 0362-1340, DOI: `10.1145/2365864.2151042`.

[14] William Enck et al., “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”, *in*: *ACM Trans. Comput. Syst.* 32.*2* (June 2014), ISSN: 0734-2071, DOI: `10.1145/2619091`.

[15] Nickolai Zeldovich et al., “Making information flow explicit in HiStar”, *in*: *Commun. ACM* 54.*11* (Nov. 2011), pp. 93–101, ISSN: 0001-0782, DOI: `10.1145/2018396.2018419`.

[16] N. Vachharajani et al., “RIFLE: An Architectural Framework for User-Centric Information-Flow Security”, *in*: *37th International Symposium on Microarchitecture (MICRO-37’04)*, 2004, pp. 243–254, DOI: `10.1109/MICRO.2004.31`.

[17] Daniel Townley et al., “LATCH: A Locality-Aware Taint CHecker”, *in*: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’52, Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 969–982, ISBN: 9781450369381, DOI: `10.1145/3352460.3358327`.

[18] Joël Porquet and Simha Sethumadhavan, “WHISK: An uncore architecture for Dynamic Information Flow Tracking in heterogeneous embedded SoCs”, *in*: *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013, pp. 1–9, DOI: `10.1109/CODES-ISSS.2013.6658991`.

[19] Michael Dalton, Hari Kannan, and Christos Kozyrakis, “Raksha: a flexible information flow architecture for software security”, *in*: *SIGARCH Comput. Archit. News* 35.*2* (June 2007), pp. 482–493, ISSN: 0163-5964, DOI: `10.1145/1273440.1250722`.

[20] Mohit Tiwari et al., “Complete information flow tracking from the gates up”, *in*: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, USA: Association for Computing Machinery, 2009, pp. 109–120, ISBN: 9781605584065, DOI: `10.1145/1508244.1508258`.

[21] Wei Hu et al., “Theoretical Fundamentals of Gate Level Information Flow Tracking”, *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.*8* (2011), pp. 1128–1140, DOI: `10.1109/TCAD.2011.2120970`.

[22] Hari Kannan, Michael Dalton, and Christos Kozyrakis, “Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor”, *in*: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009, pp. 105–114, DOI: `10.1109/DSN.2009.5270347`.

[23] Muhammad Abdul Wahab et al., “A small and adaptive coprocessor for information flow tracking in ARM SoCs”, *in*: *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2018, pp. 1–8, DOI: `10.1109/RECONFIG.2018.8641695`.

[24] Muhammad A. Wahab et al., “ARMHEx: A hardware extension for DIFT on ARM-based SoCs”, *in*: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–7, DOI: `10.23919/FPL.2017.8056767`.

[25] Shimin Chen et al., "Flexible Hardware Acceleration for Instruction-Grain Program Monitoring", *in*: *SIGARCH Comput. Archit. News* 36.*3* (June 2008), pp. 377–388, ISSN: 0163-5964, DOI: `10.1145/1394608.1382153`.

[26] Vijay Nagarajan et al., "Dynamic Information Flow Tracking on Multicores", *in*: *Workshop on Interaction between Compilers and Computer Architectures*, 2008, URL: `https://www.research.ed.ac.uk/en/publications/dynamic-information-flow-tracking-on-multicores`.

[27] Olatunji Ruwase et al., "Parallelizing dynamic information flow tracking", *in*: *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, Munich, Germany: Association for Computing Machinery, 2008, pp. 35–45, ISBN: 9781595939739, DOI: `10.1145/1378533.1378538`.

[28] Christian Palmiero et al., "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`.

[29] H. Bar-El et al., "The Sorcerer's Apprentice Guide to Fault Attacks", *in*: *Proceedings of the IEEE* (2006), DOI: `10.1109/JPROC.2005.862424`.

[30] Christian Palmiero et al., *A Hardware Dynamic Information Flow Tracking Architecture for Low-level Security on a RISC-V Core*, 2018, URL: `https://github.com/sld-columbia/riscv-dift`.

[31] Sandra Loosemore et al., *The GNU C Library Reference Manual*, 2023, URL: `https://www.gnu.org/s/libc/manual/pdf/libc.pdf`.

[32] William Pensec, Vianney Lapôtre, and Guy 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*, SensorsS&P, Istanbul, Turkiye: Association for Computing Machinery, 2023, pp. 8–14, DOI: `10.1145/3628356.3630116`.

[33] Jan Richter-Brockmann et al., "FIVER – Robust Verification of Countermeasures against Fault Injections", *in*: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), DOI: `10.46586/tches.v2021.i4.447-473`.

[34] Victor Arribas, Svetla Nikova, and Vincent Rijmen, "VerMI: Verification Tool for Masked Implementations", *in*: *25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2018, DOI: `10.1109/ICECS.2018.8617841`.

[35] Gilles Barthe et al., "maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults", *in*: *Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Proceedings, Part I*, 2019, DOI: `10.1007/978-3-030-29959-0_15`.

[36] Simon Tollec et al., "Fault-Resistant Partitioning of Secure CPUs for System Co-Verification against Faults", *in*: (2024), URL: `https://eprint.iacr.org/2024/247`.

[37] Asmita Adhikary and Ileana Buhan, "SoK: Assisted Fault Simulation", *in*: *Applied Cryptography and Network Security Workshops*, Springer Nature Switzerland, 2023, DOI: `10.1007/978-3-031-41181-6_10`.

[38] Riscure, *FiSim: An open-source deterministic Fault Attack Simulator Prototype*, URL: `https://github.com/Riscure/FiSim`.

[39] Victor Arribas et al., "Cryptographic Fault Diagnosis using VerFI", *in*: *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, DOI: `10.1109/HOST45689.2020.9300264`.

[40] Gaetan Canivet et al., "Glitch and laser fault attacks onto a secure AES implementation on a SRAM-based FPGA", *in*: *Journal of cryptology* (2011), DOI: `10.1007/s00145-010-9083-9`.

[41] Florian Hauschild et al., "ARCHIE: A QEMU-Based Framework for Architecture-Independent Evaluation of Faults", *in*: *Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 2021, DOI: `10.1109/FDTC53659.2021.00013`.

[42] Yohannes B. Bekele, Daniel B. Limbrick, and John C. Kelly, "A Survey of QEMU-Based Fault Injection Tools & Techniques for Emulating Physical Faults", *in*: *IEEE Access* (2023), DOI: `10.1109/ACCESS.2023.3287503`.

[43] Ralph Nyberg et al., "Closing the Gap between Speed and Configurability of Multi-bit Fault Emulation Environments for Security and Safety-Critical Designs", *in*: *17th Euromicro Conference on Digital System Design*, 2014, DOI: `10.1109/DSD.2014.39`.

[44] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini, "Shaping the Glitch: Optimizing Voltage Fault Injection Attacks", *in*: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), DOI: `10.13154/tches.v2019.i2.199-224`.

[45] Paul Grandamme, Lilian Bossuet, and Jean-Max Dutertre, "X-Ray Fault Injection in Non-Volatile Memories on Power OFF Devices", *in*: *2023 IEEE Physical Assurance and Inspection of Electronics (PAINE)*, 2023, DOI: `10.1109/PAINE58317.2023.10318018`.

[46] Brice Colombier et al., "Multi-Spot Laser Fault Injection Setup: New Possibilities for Fault Injection Attacks", *in*: *Smart Card Research and Advanced Applications*, 2022, DOI: `10.1007/978-3-030-97348-3_9`.

[47] Tobias Schneider, Amir Moradi, and Tim Güneysu, "ParTI–towards combined hardware countermeasures against side-channel and fault-injection attacks", *in*: *Advances in Cryptology–CRYPTO: 36th Annual International Cryptology Conference, Proceedings, Part II 36*, 2016, DOI: `10.1007/978-3-662-53008-5_11`.

[48] William Pensec, *FISSA: Fault Injection Simulation for Security Assessment*, URL: `https://github.com/WilliamPsc/FISSA`.

[49] Siemens, *QuestaSim*, URL: `https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/`.

[50] Verilator, *Verilator*, URL: `https://github.com/verilator/verilator`.

[51] Xilinx, *Vivado Design Suite*, URL: `https://www.xilinx.com/products/design-tools/vivado.html`.

[52] Michael L. Waskom, "Seaborn: statistical data visualization", *in*: *Journal of Open Source Software* (2021), DOI: `10.21105/joss.03021`.

[53] J. D. Hunter, “Matplotlib: A 2D graphics environment”, *in*: *Computing in Science & Engineering* (2007), DOI: `10.5281/zenodo.7697899`.

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.