COLLEGE MATHSTIC
DOCTORAL BRETAGNE
BRETAGNE OCEANE

Université Bretagne Sud

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE BRETAGNE SUD

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

Par
William PENSEC

Amélioration de la Protection des Processeurs Contre des Menaces Logicielles et Physiques par la Sécurisation d'un Mécanisme de Sécurité DIFT Contre des Attaques par Injections de Fautes

Enhanced Processor Defence Against Physical and Software Threats by Securing DIFT Versus Fault Injection Attacks

Thèse présentée et soutenue à Lorient, le 19/12/2024
Unité de recherche : Université Bretagne Sud, UMR CNRS 6285, Lab-STICC
Thèse N° : « si pertinent »

Rapporteurs avant soutenance :

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

Composition du Jury :

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

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

Invité(s) :

Prénom NOM Fonction et établissement d'exercice

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

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

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

Remerciements

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

RÉSUMÉ

Abstract

Embedded systems are increasingly prevalent in critical infrastructures such as industries, smart cities, and biomedical devices, improving efficiency and addressing challenges like climate change and health. However, their widespread use also expands the attack surface, creating significant security risks. These systems, typically powered by low-energy processors handling sensitive data, are vulnerable to both software and physical attacks due to their network connectivity and proximity to potential attackers. Hence, addressing both threats during processor designing is essential.

Dynamic Information Flow Tracking (DIFT) techniques, which detect software attacks like buffer overflow and malware by attaching and propagating tags to data at runtime, are a key defence. Fault Injection Attacks (FIA) deliberately induce errors in a system's hardware to alter its normal operation, often bypassing security mechanisms. These faults can be introduced via physical methods (e.g., voltage, lasers), leading to potential data breaches or system disruptions. FIAs are particularly concerning in embedded systems and cryptographic devices, where low-level faults can compromise sensitive information. Many studies have shown different vulnerabilities due to FIAs on critical systems, but none of them targetted a DIFT mechanism.

We focus on the D-RI5CY processor, which implements a hardware-based in-core DIFT. Our primary objective is to assess the impact of FIA on the effectiveness of DIFT in the D-RI5CY processor. Through fault injection simulations, we evaluate the vulnerability of DIFT and identify critical hardware components requiring protection. As a result of this evaluation, we implemented two lightweight countermeasures, considering constraints like area and performance: simple parity for error detection and Hamming Code for single-bit error detection and correction. These were optimised by grouping registers to reduce parity/redundancy overhead. The sensitivity evaluation was conducted using FISSA, a tool developed during this PhD work to facilitate fault evaluation at the conceptual stage. This tool allows the enabling of the principle of *Security by Design*. Finally, we evaluated the security of multiple register group compositions to enhance countermeasure effectiveness against complex fault models. We tested Hamming Code with five group configurations and developed a new version of the code capable of detecting two errors and correcting one (SECDED). This was compared across the same groups in terms of efficiency and area to find the optimal trade-off for embedded systems with strict energy and performance constraints.

RÉSUMÉ ÉTENDU

Table of Contents

Acronyms

AES Advanced Encryption Standard

ALU Arithmetic and Logical Unit

API Application Programming Interface

CABA Cycle Accurate and Bit Accurate

CPU Central Processing Unit

CSR Control and Status Registers

DDoS Distributed Denial of Service

DIFT Dynamic Information Flow Tracking

DUT Device Under Test

ECC Error Correcting Code

EDC Error Detecting Code

EM Electromagnetic

EMFI Electromagnetic Fault Injection

FF Flip-Flop

FIA Fault Injection Attack

FPGA Field Programmable Gate Array

GUI Graphical User Interface

HDL Hardware Description Language

IC Integrated Circuit

IIoT Industrial Internet of Things

IoT Internet of Things

ISA Instruction Set Architecture

LUT Look-Up Table

MMU Memory Management Unit

OS Operating System

PC Program Counter

RA Return Address

SCA Side Channel Analysis

SECDED Single Error Correction, Double Error Detection

SoC System on Chip

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

An embedded system is a specialised computing system designed to perform dedicated functions or tasks within a larger mechanical or electrical system. Unlike general-purpose computers, embedded systems are optimised for specific control operations and are typically integrated into the hardware they manage. These systems are characterised by their compact size, low power consumption, and real-time performance constraints. They consist of microcontrollers or microprocessors, along with memory and input/output interfaces, tailored to meet the precise requirements of the application they serve. Embedded systems are ubiquitous in modern technology, powering a wide range of devices from household appliances and medical equipment to industrial machines and automotive systems, ensuring efficiency, reliability, and functionality in their operations.

The Internet of Things (IoT) has revolutionised the way we interact with technology, enabling seamless connectivity and communication between a myriad of devices. These devices are part of our daily lives, from the connected light bulb to autonomous cars. These devices collect and share data about how they are used and the environment in which they operate. Immense amounts of data are also being generated by connected cars, production, and transport applications. Today, Industrial IoT (IIoT) represents the largest and fastest-growing volume of data. To capture data, they rely on sensors embedded in every physical device, such as mobile phones, smartwatches, medical devices (pacemakers, cardiac defibrillators, etc.), but also in recent cars, or in agriculture

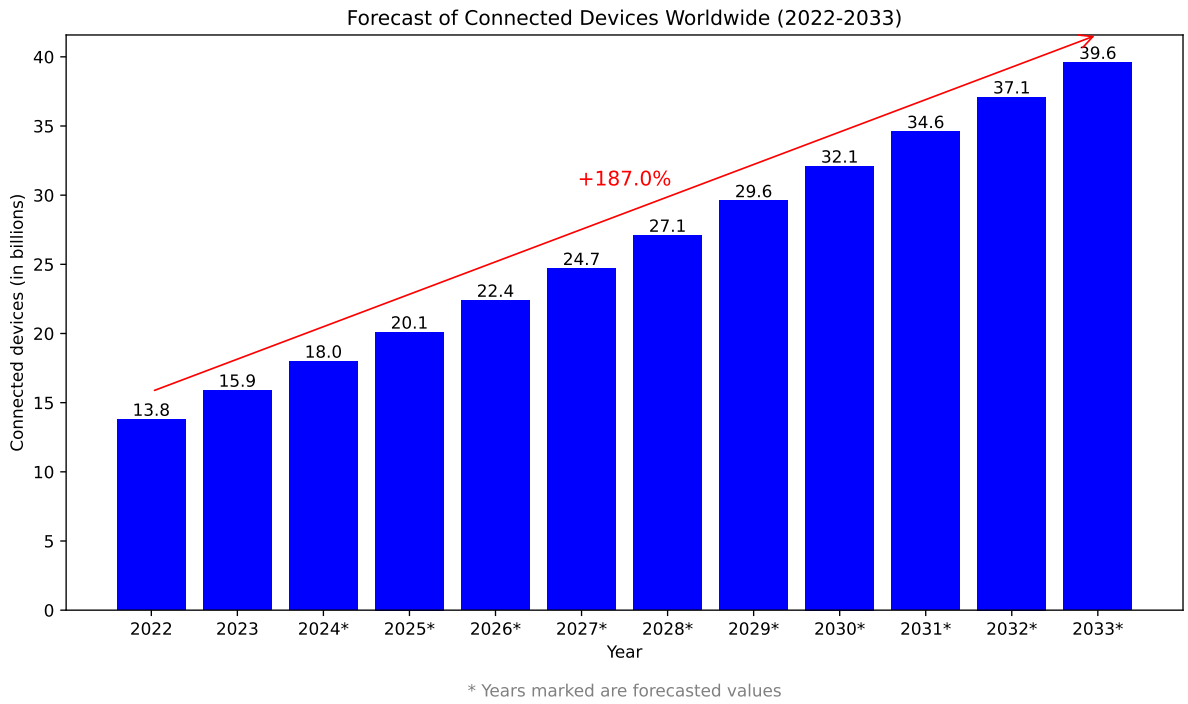


Figure 1.1: Number of IoT (IoT) devices worldwide from 2022 to 2033 (from [1])

to monitor humidity, temperature, or automate the irrigation system. These sensors generate data that can be critical, and as these data exist, they are subjects of cyber-attacks. According to forecasts, the number of IoT devices in use worldwide is estimated to reach approximatively 40 billion in 2033 [1], as shown in Figure 1.1, while, today, in 2024, we count around 18 billion. The economic impact of IoT is substantial, with worldwide consumer IoT revenue expected to rise from $181.5 billion in 2020 to $621.6 billion by 2030 [2] as shown in Figure 1.2. As IoT continues to expand its reach, the importance of ensuring robust security in these systems becomes increasingly critical. IoT devices, often characterised by limited resources and large-scale deployment, present unique security and privacy challenges.

Embedded systems, which form the backbone of IoT devices, are increasingly vulnerable to both software and hardware threats, as well as network-based threats, which can lead to data leaks or unauthorised access to essential system components. These systems are frequently deployed in environments where they are exposed to potential adversaries, making them attractive targets for various types of attack [3, 4].

Software security is a critical aspect of the development and deployment of software systems, encompassing measures and practices designed to protect applications from malicious attacks, vulnerabilities, and other security risks. It involves the implementation of protocols to ensure the confidentiality, integrity, and availability of software and data. This field addresses a wide

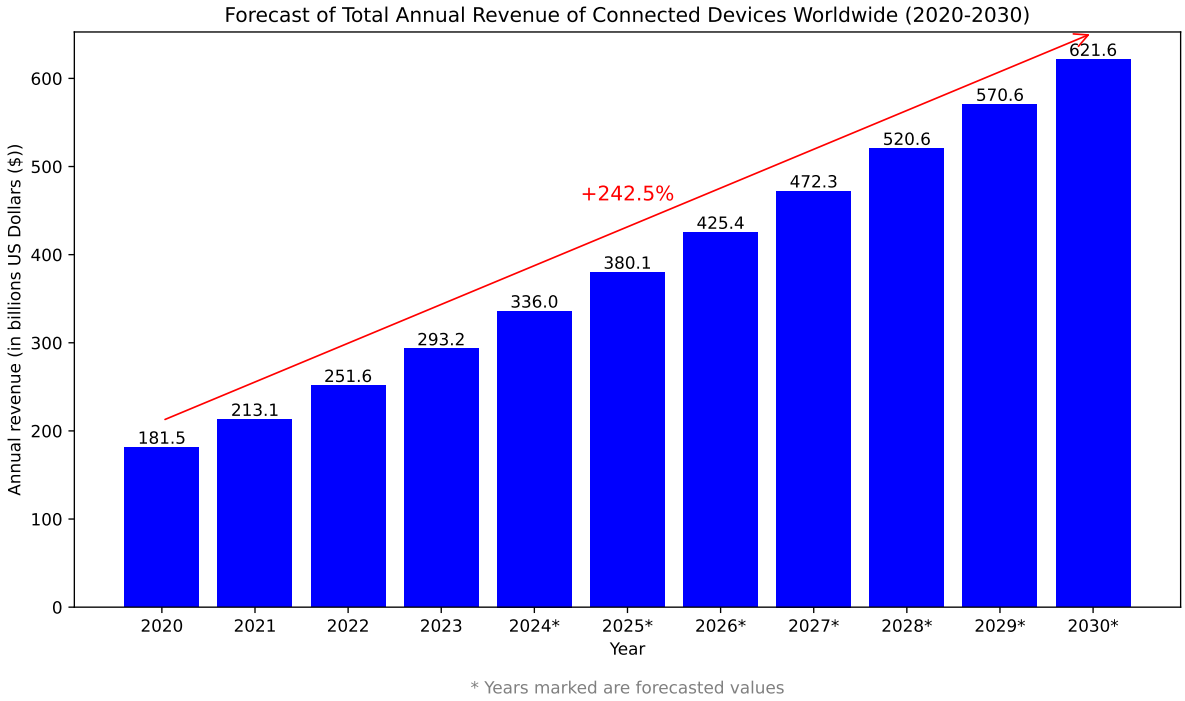


Figure 1.2: Internet of Things total annual revenue worldwide from 2020 to 2030 (from [2])

range of threats, including but not limited to, malware [5], memory overflow attacks [6], SQL injection [7], and Cross-site scripting (XSS) [8]. Effective software security practices include rigorous code reviews, the use of secure coding standards, regular vulnerability assessments, and the deployment of encryption and authentication mechanisms. As software becomes increasingly integral to various aspects of daily life and business operations, ensuring its security is paramount to safeguarding sensitive information, maintaining user trust, and preventing financial and reputational damage.

Network attacks, such as Distributed Denial of Service (DDoS) attacks, can overwhelm an embedded system's network interface, rendering it inoperative, while man-in-the-middle attacks [9] intercept and potentially alter communication between devices, Internet Protocol spoofing [10], jamming [11], and many others. These vulnerabilities can be exploited to leak confidential data, corrupt system functionality, or gain control over critical system operations, underscoring the urgent need for robust security mechanisms in embedded systems.

On the hardware front, physical attacks refer to different techniques and methods aimed at compromising the security of embedded systems. These attacks exploit vulnerabilities in the physical layer or implementation of the device's hardware to delete, modify, gain or prevent access to confidential data. The most common physical attacks are reverse engineering, Side-Channel Attacks (SCA) and Fault Injection Attacks (FIA).

Side-channel attacks [12] are passive physical attacks that primarily aim to exploit leakages of information from a device, such as power consumption, electromagnetic emissions, or timing information. By capturing and analysing these side-channel data, attackers can infer sensitive information, such as cryptographic keys [13].

Fault injection attacks [14–16] are active physical attacks, noninvasive or invasive, transient or permanent, where the attacker intentionally try to change the normal behaviour of a device during program execution by injecting one or more faults, then observing the erroneous behaviour that could be further exploited as a vulnerability. Boneh et al. [17] introduced fault injection attacks. They were able to break some cryptographic protocols by inducing faults into the computations.

In this dissertation, we only study and present fault injection attacks. Nowadays, these attacks are more and more easier to make. For example, NetSPI introduced, in the Black Hat conference in Las Vegas, in August 2024, a new laser hacking device called the RayV Lite [18]. The authors, Sam Beaumont and Larry "Patch" Trowell, presented their open-source tool that aims to let anyone achieve laser-based tricks to reverse engineer chips and trigger their vulnerabilities. There are already some tools such as Riscure Laser Station [19] who costs between $10,000 and $150,000. In the same way as NewAE [20, 21] with their ChipWhisperer or ChipShouter that allow to realise clock glitching, voltage glitching or even electromagnetic injection at a lower cost and more accessible, RayV Lite allows people to perform laser-based attacks for only $500 which is more accessible and cheaper than any other tools available. Another work, in 2020, from M. S. Kelly and K. Mayes [22] presented a setup with cheap components where they were able to make a laser setup for around $500. The low cost and relative ease of construction of their laser environment suggests that developers of IoT devices need to seriously consider this threat on their devices, because it must be assumed that these attack techniques are readily available to malicious attackers.

Many studies have shown the vulnerabilities of critical systems against FIAs. [23] demonstrates that it is possible to recover computed secret data using FIA in hidden registers on the RISC-V Rocket processor. Electromagnetic fault injection (EMFI) attack can be used to recover an AES key by targeting the cache hierarchy and the MMU, as shown in [24]. Laser fault injections (LFI) can allow the replay of instructions [25], that can lead to the overwriting of an entire section of a program. [26] shows the use of glitch injections on the power supply to control the program counter (PC). Voltage glitches can also lead to glitch TrustZone mechanisms, as shown in [27]. Finally, authors of [28] have shown that one can combine side-channel attacks (SCA) and FIAs to bypass the PMP mechanism in a RISC-V processor.

Thus, the main research question of this work is how can we maintain maximum protection against software attacks in the presence of physical attacks ?

1.2 Objectives

In this dissertation, we address a part of the threats that IoT devices faces, with a particular emphasis on security threats affecting the software and hardware layers of a device. The main objective is to provide a robust security mechanism against both software and physical threats, where the attacker performs a fault injection attack to bypass a software security mechanism in order to realise a software attack. We rely on a security mechanism called Dynamic Information Flow Tracking (DIFT) to protect the system against software attacks. This mechanism is presented in Chapter 2.2.

The first contribution of this dissertation is to show that this mechanism is vulnerable to fault injection attacks, using an HDL simulator tool to simulate the behaviour of a processor in the presence of fault injections targeting the DIFT mechanism at runtime.

The second contribution is the development of a tool for automating the simulation process on a given processor design. This open-source tool is available on GitHub and can be used during the development process to find the vulnerabilities of an HDL design. Thanks to this tool, the designer is able to check his design right from the conceptual phase in order to have a robust design against fault injection attacks, enabling the notion of *Security by Design.*

The third contribution is the implementation of two lightweight countermeasures inside the DIFT mechanism to protect it against fault injection attacks. For the countermeasures, we take into account various constraints such as area, and performance overhead.

Finally, in our last contribution, we evaluate different implementations of lightweight countermeasures to protect the mechanism against stronger fault model.

1.3 Manuscript outline

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

Chapter 2 presents the state of the art of this dissertation and define the different technical terms. Firstly, it presents Information Flow Tracking (IFT), and its different types. Secondly, this chapter presents physical attacks, focusing on the three mains types: Reverse Engineering, Side-Channel Attacks and Fault Injection Attacks. Finally, the chapter presents an overview of the literature about countermeasures against Fault Injection Attacks, and provides a small discussion on their advantage and disadvantages.

Chapter 3 presents the background of this work with the presentation of the RISC-V Instruction Set Architecture (ISA), the architecture of the D-RI5CY core in detail. Then, the different use cases are presented in details, highlighting their software vulnerability which can be detected by a DIFT mechanism. Finally, a vulnerability assessment is done to show how the considered DIFT mechanism is vulnerable against FIA in these examples 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 its software architecture and how to use it, and compares it to others tools available in the literature.

Chapter 5 details the different implementation of lightweight countermeasures to protect the D-RI5CY core against FIA, taking into account common fault models. Then we evaluate these protections in terms of area, performance, and efficiency.

Chapter 6 evaluate the countermeasures performances against more complex fault models, and we introduce a third countermeasure against more complex fault models. Then we evaluate these protections in terms of area, performance, and efficiency.

Chapter 7 is dedicated to the summary of this dissertation with a short discussion on the obtained results, identifying limitations, and discussing the challenges encountered in this thesis. We also explore future research perspectives at short and long terms, and suggest potential improvements.

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 discuss the various levels of monitoring, from program behaviour to the detection of hardware trojans. Then in Section 2.3, we provide an overview of the different existing physical attacks, focusing on Fault Injection Attacks (FIA). Finally in Section 2.4, we present existing 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 La-Padula [29] and by Denning [30] 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 How hardware DIFT work

DIFT is a technique used in computer security to monitor the flow of information through a system. It aims to prevent security breaches such as data leaks, unauthorised data manipulation, and execution of untrusted code. In DIFT, each data is associated with a tag that indicates its security level. For example, a tag might indicate whether data is 'trusted' or 'untrusted'. When data is input into the system, it is initially tagged based on its source.

As data moves through the system, these tags are tracked to ensure compliance with security policies and to ensure that sensitive information does not get exposed or manipulated improperly. For instance, if an operation involves both trusted and untrusted data, the result might be tagged as untrusted to ensure security.

An example of such security policy can be represented in Table 2.1. In this example, if the data comes from the network or if it's manipulated by a user, in the case of a `scanf()` function in C language for example, the data cannot be trusted, while if the data comes from a secure channel or is manipulated by the system itself, the data can be trusted.

Table 2.1: Security policies for different data inputs

Data Input	**Security Policy**	**Tag**
User Input	User-provided	Untrusted
Network	External source	Untrusted
Internal	System-provided	Trusted

Figure 2.1 illustrates the three main steps of how DIFT works. Firstly, three data, D_1, D_2, and D_3, with their associated tags in two different colours, are initialised on the left side of the figure. In the second step, when the data are fetched by the core for computation, the associated tags are propagated inside the core and confronted with the propagation policy depending on the operations performed on the data. Finally, in the last step, on the right side of the figure, there are two data outputs derived from the three initial data. Data D_4 results from the combination of data D_2 and D_3, while data D_5 is derived only from data D_1. Since data D_1 has not been modified, its tag remains the same. However, the tag associated to D_4 is the combination of tags from D_2 and D_3. Depending on the security policy, if D_3 was trusted and D_2 was not, the output tag will be *untrusted* (i.e, as in the Figure). Consequently, when the tags go through the final step of DIFT, they will be checked, and an exception may be raised, or the application may be stopped due to the combination of *trusted* and *untrusted* values.

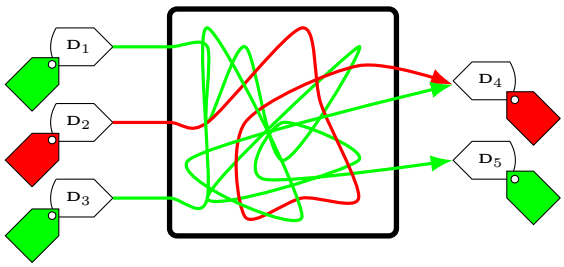


Figure 2.1: Representation of the DIFT mechanism from initialisation to checking.

2.2.2 Different types of IFT

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

2.2.2.1 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 [31]. 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 [32], such as verifying secure information flow in operating systems, programming languages with built-in information flow controls, and hardware design for secure systems [33].

2.2.2.2 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 [34]. DIFT operates by tagging or labelling input data from potentially untrusted sources and tracking how this data propagates through the system [35]. 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 [34]. DIFT has proven effective in detecting and preventing a wide range of security vulnerabilities, including buffer overflows, format string attacks, and code injection attacks [35]. 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.

2.2.3 Different levels of DIFT

IFT can be implemented at various levels of abstraction in computing systems [31, 34, 36]. 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 2.2 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 [37]. 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 commu-

nicate 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.

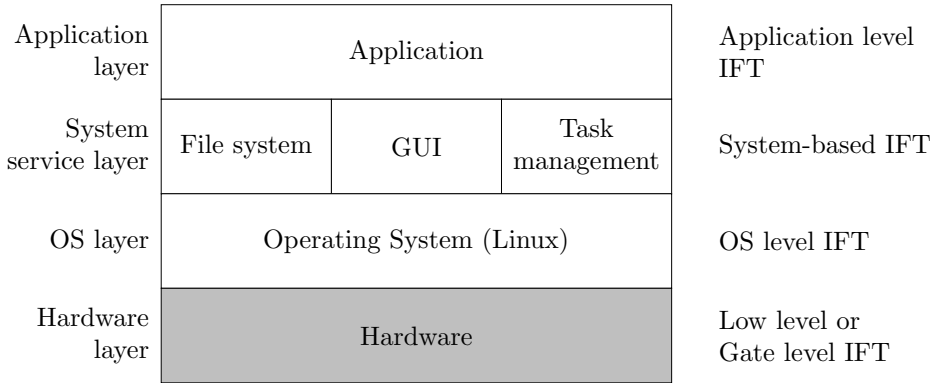


Figure 2.2: Simplified representation of the different layers in an embedded system

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 explore these different levels, highlighting their respective benefits and limitations.

2.2.3.1 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 program-

ming languages. JFlow [38] 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 [39, 40]. Austin et al. [40] 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. [41] 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 [42] 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 tag 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 [43] is an OS that has been designed to provide precise data specific security policies. The authors propose to assign tags to different objects in the operating system instead of data.

2.2.3.2 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 [34, 36].

One example of this type of DIFT is RIFLE [44], 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 works with every programs that run on a system, and policy decisions are left to the user, not the programmer. Townley et al. [45] 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. [46] 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 the exokernel-based MutekH operating system and provides support for tag page allocation, configuration of the page table cache, and interrupt management for writing to untagged pages.

2.2.3.3 Hardware-based DIFT

Dalton et al. [47] 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 [31, 36] 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.

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 core. Processor manufacturers do not prioritise this type of approach, and as most processors are not open to the public, it is difficult to modify them.

Kannan et al. [48] 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 2.3 represents how an off-core DIFT would be implemented. Kannan et al. uses this idea for implementing their solution. This coprocessor handles all DIFT functionalities, synchronizing with the main processor only during system calls. This design eliminates the need for changes to the main processor's pipeline, logic, or caches, making the solution 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 coproces-

sor provides the same security guarantees as in-core 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.

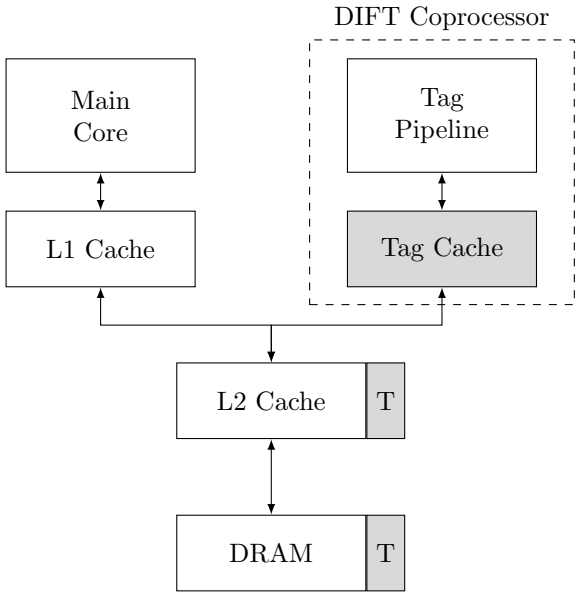


Figure 2.3: Representation of a Hardware Off-Core DIFT (inspired by Figure 1 of [48])

Wahab et al. [49, 50] developed 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 have 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, [49] gives around 5% communication overhead and an area overhead of 0.47% from the baseline CPU, i.e. Cortex-A9 without a DIFT, and a power consumption increased by 16%; while [50] gives a communication overhead of 335%, an area increased by 0.95% and a power consumption increased by 16.2%.

Off-Loading DIFT uses a dedicated core of a multicore CPU [51–53]. Figure 2.4 represents Off-Loading DIFT principle with a core running the application and another, in parallel, run the DIFT analysis on the application trace. The application core is instrumented in order to generate a trace and compress it. The trace includes executed instructions and packs 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. 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 multicore CPU, reducing the number of core available and increasing the energy consumption due to the application trace analysis. In an embedded system where consumption is a critical factor, this solution is difficult to consider.

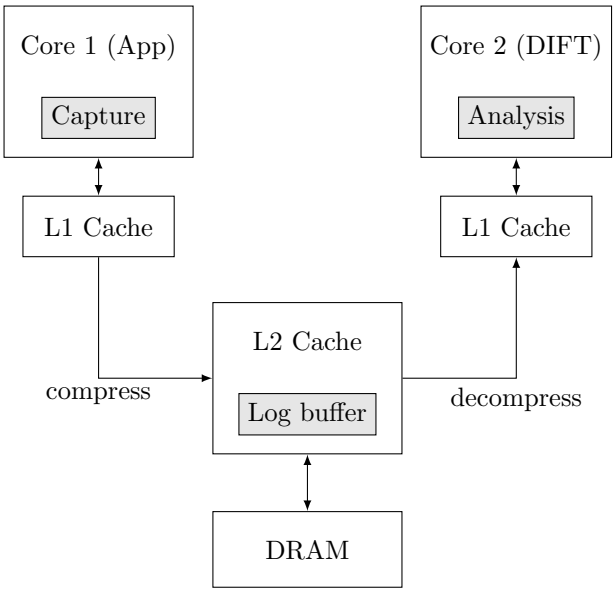


Figure 2.4: Representation of a Hardware Off-Loading DIFT (inspired by Figure 1 of [48])

In-Core DIFT relies on a deeply modified processor pipeline which needs 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 intercore 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, a tag computation unit, and first level of caches must be added to store tags in parallel with the regular blocks into the processor core. Figure 2.5 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 conforms to the security policy, the tag, and the ALU output are saved into the Tag Register File, or possibly, stored in memory. Otherwise, if the tag does not conform, the DIFT mechanism

detects the security violation and can raise an exception. The DIFT reaction policy is not an integral part of DIFT but depends on the higher-level OS or software.

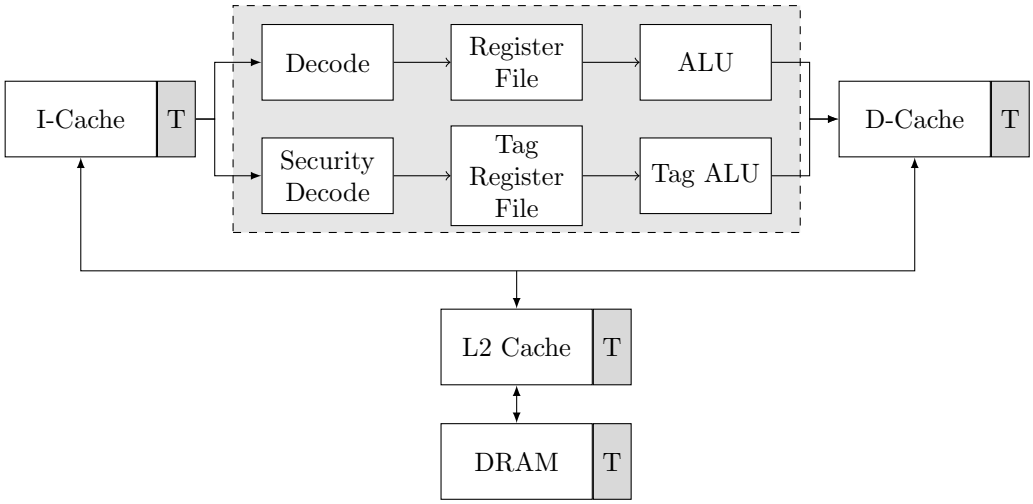


Figure 2.5: Representation of a Hardware In-Core DIFT (inspired by Figure 1 of [48])

Suh et al. [35] 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 can detect various threats, such as attacks targeting instructions or jump addresses. If the security policy detects something malicious in hardware, the OS will process the exception. They use a 1-bit tag, which means only two ways of representing 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 the first policy.

Dalton et al.[47] 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. 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 high processor privilege (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 implemented onto an FPGA board.

Palmiero et al. [54] 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 [47], Palmiero et al. [54] 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. This solution is described in detail in Chapter refsection:driscy.

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

GLIFT [55] is a well-established IFT technique. The goal is to protect against hardware trojans and unauthorized behaviours. 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. [56] 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 fan-outs. Many other works have been done on GLIFT to attempt a decrease of the logic complexity.

2.3 Physical Attacks

This section presents an overview of the state of the art on physical attacks. We introduce the different types of physical attacks and their methods to recover secret information. Firstly, we begin with Reverse Engineering, how to retrieve information from a product to recover useful information. Secondly, we begin with Side-Channel Attacks, how to use information leakage to recover useful information and how to analyse them. Finally, we introduce Fault Injection Attacks. We define the different possibilities of injection and how to achieve them.

2.3.1 Reverse Engineering

Reverse engineering refers to the process of information retrieval from a product, ranging from aircraft to modern Integrated Circuits (IC). Reverse engineering of IC is a complex process that involves analysing and understanding the design, functionality, and operation of existing hardware. This technique is used for various purposes in the electronics industry, such as to gain a full understanding of its construction and or functionality [58]. To reverse engineering

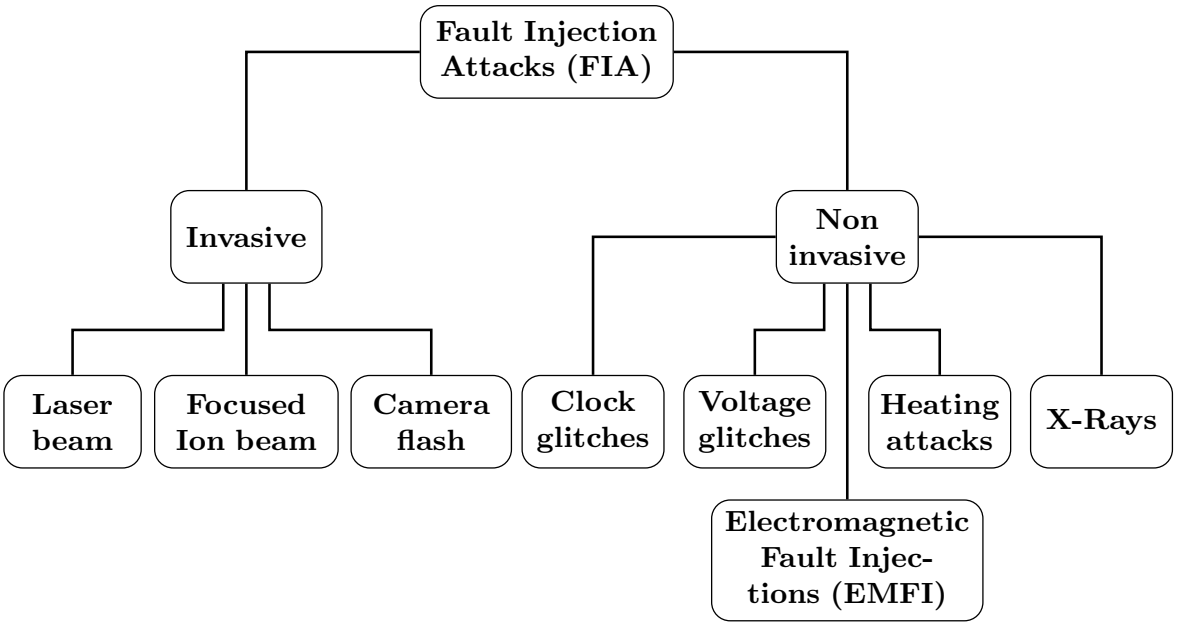


Figure 2.6: Taxonomy of the different methods of physical attacks (inspired by [57])

a chip [59], an attacker need to remove the chip protection in order to observe it thanks to a scanning electron microscope (SEM) or another method Focused Ion Beam (this method is explained in Chapter 2.3.3.1). Also knowing the region of interest is beneficial as the planar surface can be reduced significantly.

2.3.2 Side-Channel Attacks

Side-Channel Attacks exploit information leakages on the circuit behaviour such as power consumption, electromagnetic (EM) radiation or the execution time of an application. This type of attack does not call into question the theoretical integrity of the target algorithm, but aims to recover information by devious means due to its implementation. During data processing, the alternation between different states requires time and minimal energy dissipation, the variations of which can be analysed by the attacker. This information allows the attacker to access secret data such as a password, or cryptographic key. The origin of these attacks date back to the TEMPEST program from NSA [60]. They described the vulnerabilities of a cryptographic implementation from their electromagnetic emissions, depending on the input and data.

Figure 2.7 represents the different methods of SCA on a microprocessor. The main idea is to have an application running on the processor and an attacker will use one method to trace the application multiple times to recover secret information (e.g. cryptographic key, password, private data, etc.).

Multiples possibilities exist to exploit SCA. As seen on Figure 2.7, power analysis exploits

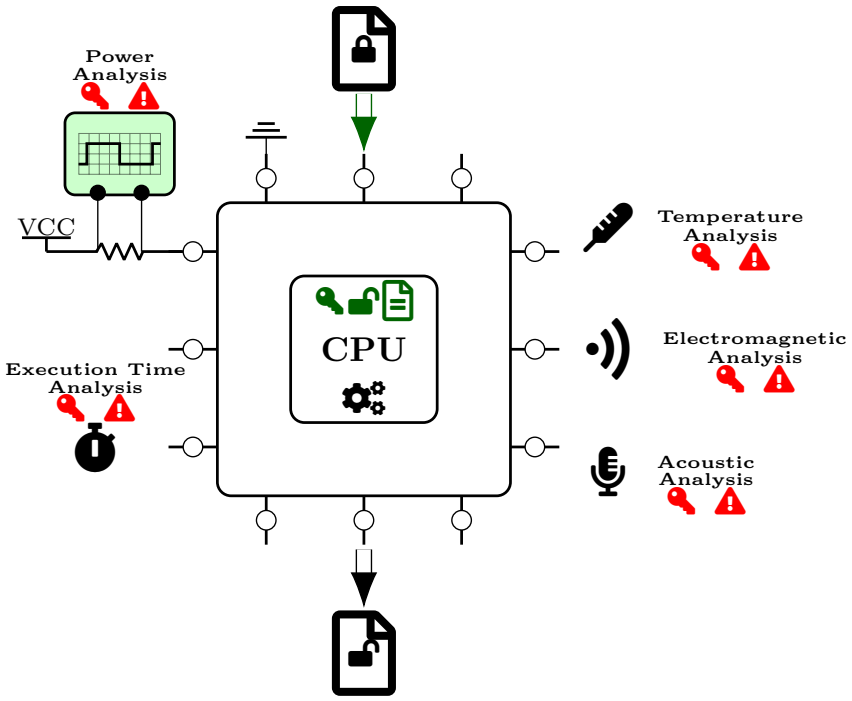


Figure 2.7: Representation of the different methods of Side-Channel attacks

time differences in target power consumption during sensitive executions. Modern systems contain billions of transistors (up to 208 billions transistors for an Nvidia GPU GB200 Grace Blackwell[1]). These transistors act as voltage switches and as they are continually switched on/off during execution, they cause voltage variations that can be observed and measured using equipments and devices (oscilloscope, voltmeter, etc.). These results data are analysed and from a certain number of data, an attacker can deduce secrets [61–64].

Another possibility is to analyse the execution time of a program also known as timing attacks and first introduced by Kocher [13], takes advantage of the fact that some sensitive computational operations vary in time depending on their secret inputs [65]. A third possibility is to exploit electromagnetic (EM) [66–70] emissions signatures produced when conducting logic operations. Thus, EM emissions reflect the operations of the system. In 2001, Quisquater and Samyde [71] extended SCA with EM analysis. Another method is to analyse to exploit the temperature [72, 73] values induced by the activity of the system. This method is linked to EM emissions and power analysis, as they use traces from the system's execution. Finally, last but not least, an attacker could use acoustic analysis [74–76] to extract confidential secret from the sound emitted by the system. This technology has been around for a long time and is used in

1. `https://nvidianews.nvidia.com/news/nvidia-blackwell-platform-arrives-to-power-a-new-era-of-computing`

many fields, such as sonar when the system is a submarine, a warship, or a ship to distinguish one from another.

2.3.3 Fault Injection Attacks

As early as the 1970s, with advances in the space industry, anomalies in the operation of electronic circuits were observed and possibly linked to cosmic radiation outside the Earth's atmosphere [77–79]. These disturbances were initially found to affect the performance of electronic systems in space environments, where high-energy particles could disrupt the normal functioning of circuits. However, as transistors became smaller and required less energy to operate, similar phenomena were observed in terrestrial environments and aircraft systems. These transient disturbances, commonly referred to as "*soft errors*", are now recognised as a critical issue in both space and ground electronics, affecting everything from memory chips to complex processors. Figure 2.6 shows a representation of a taxonomy to classify the different method of physical attacks. Each type of attacks will be explained in the following.

However, in addition to these induces cosmic faults, wanted faults exist and are known as Fault Injection Attacks (FIA). FIA involves deliberately introducing a fault into the system to observe its behaviour and identify potential vulnerabilities. If the error caused by the fault does not propagate and execution of the application completes normally, the fault is ineffective. On the other hand, if the fault affects the execution of the application, causing it to fail or behave differently than expected, then the fault is effective. These faults can impact the performance, functionality, and reliability of the circuit. These attacks can induce errors in internal electronic components, which can be utilised to recover cryptographic keys and other secret data. These attacks have been vastly studied since the first introduction of these attacks by Boneh et al. in 1997 [17, 80]. Multiple studies or surveys [14, 16, 57, 81–83] present the different sources of FIA. Figure 2.8 presents a summary of the different methods of FIA, the figure does not represent all possible methods. Each of these attacks requires equipment which is more or less expensive and easy to acquire, ranging from a few hundred euros (clock glitches, voltage glitches) to several hundred thousand (Laser, X-Ray, Focused Ion Beam).

As shown in the Figure 2.6, these attacks are categorised as transient or permanent, and invasive or non-invasive. The effect of a transient fault lasts for a limited period of time. These faults rarely do any lasting damage to the component affected, although they can induce an erroneous state in the system. Their aim is to temporarily disrupt the program control flow or corrupt the results of an instruction to gain unauthorised access to sensitive code and data. By opposition, permanent faults or destructive faults, created by purposely inflicted defects to the chip's structure, have a permanent effect. Once inflicted, such destructions will affect the chip's behaviour permanently and persist irrespective of device restarts and resets.

Invasive attacks involve major alteration to the Device Under Test (DUT), such as decapping

the System-on-Chip (SoC) to expose its internals and remove any protective layers. These processes risk irreparable damage or destruction of the target under evaluation, potentially leading to permanent data loss.

Non-invasive attacks require no tampering of the DUT. They are able to mask their presence as they have no effect on the system other than the faults they inject.

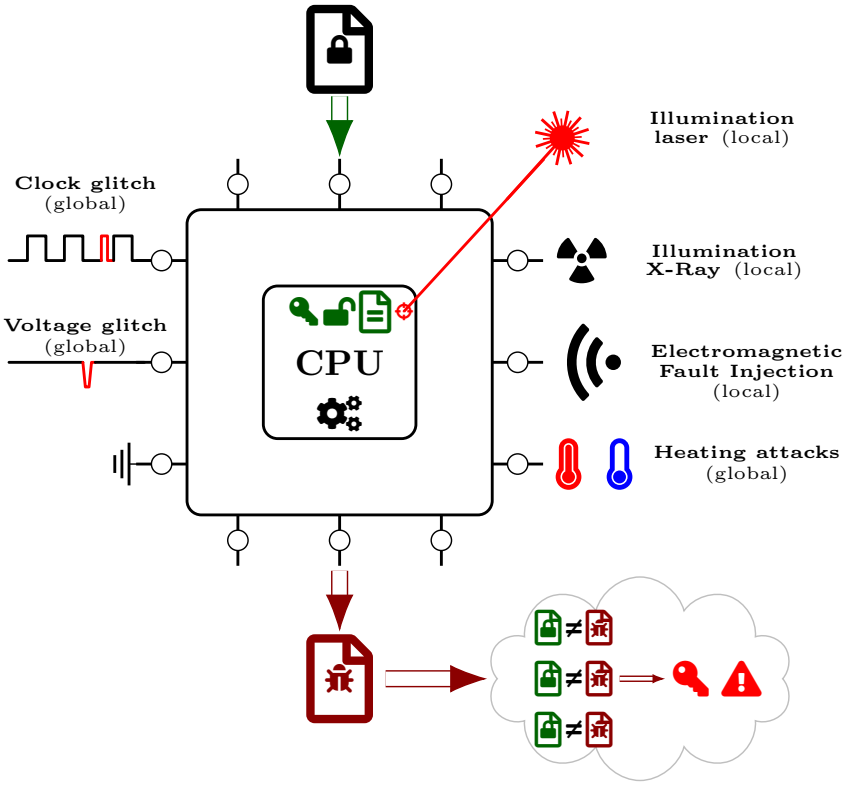


Figure 2.8: Representation of the different methods of Fault Injection attacks

2.3.3.1 Invasive attacks

Invasive attacks need to decapsulate the chip or the Integrated Circuit (IC). Decapsulating a die or an IC is a process used to expose the internal components of an IC, typically for failure analysis or reverse engineering. The goal is to carefully remove the protective encapsulation, which shields the silicon die and is typically made of epoxy or ceramic, without causing damage to the internal structures. There are several methods to achieve this, each suited to different packaging materials and levels of precision, ranging from chemical processes to advanced techniques like laser ablation and plasma etching.

The most common method is chemical decapsulation, which involves etching away the epoxy with concentrated acids such as nitric or sulphuric acid. This process requires safety precautions

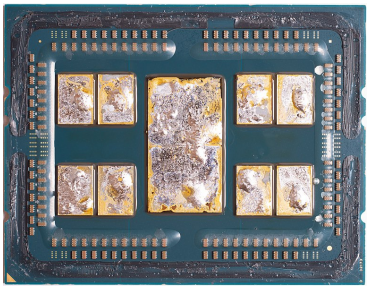
such as protective clothing and neutralisation of the acids after removal of the encapsulation. It is an effective but dangerous process and require careful control to avoid damaging the die, as over-etching can cause irreversible harm.

Another method is laser decapsulation, which uses a precision laser to remove the encapsulation material layer by layer. This technique is highly accurate and reduces the risk of damage to the die, but it is expensive and requires specialised equipment. Mechanical decapsulation involves physically grinding or cutting away the encapsulation, but has a high risk of damaging the die, especially when approaching the final layers.

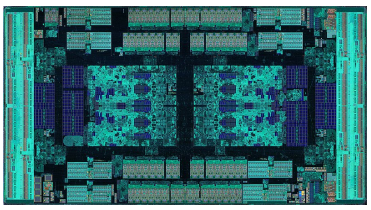
Plasma etching is a more advanced technique that uses ionised gases to gradually etch away the encapsulation material. It offers high precision but is slower than other methods and is typically used in research or industrial environments. Whichever method is used, safety precautions are essential, especially when dealing with hazardous chemicals and sensitive materials.

Figure 2.9 shows three different steps to decapsulate a circuit. To be noted, this processor is the AMD Zen2 EPYC 7702 server processor, which is not for embedded systems.

(a) Initial die from an AMD Zen 2 EPYC 7702 server processor.



(b) AMD EPYC 7702 after CPU delidding.



(c) Die shot of the centre die, after removal from processor package.

Figure 2.9: Three steps to decapsulate a die (from [84])

Camera flash/light source is a type of optical attack. The attacker needs to decapsulate the chip, and the strong radiation emitted by the flash directed at the silicon surface can cause the blanking of memory cells where constant values are stored for algorithms execution (e.g., the AES S-Boxes). These attacks are inexpensive, but, in the other hand, they are not very accurate. Skorobogatov et al. [85] used a flashgun for $30 while being able to change any bit of an SRAM array.

Schmidt and Hutter [86] present practical attacks on implementations of RSA that use Chinese Remainder Theorem (CRT). These attacks have been performed into a cryptographic device through optical and EM injections. They use a laser diode as a light source, the diode emits a light beam of 100 mW with a wavelength of 785 nm. The light from the diode is guided thanks to a fibre-optic of 1 mm in diameter. Guillen et al. [87] present a low-cost fault injection

setup, around a couple of hundred euros, which is capable of producing localized faults in modern 8-bit and 32-bit microcontrollers. This setup does not require handling dangerous substances or wearing protection equipment. The fault produced by this setup are able to successfully attack real-world cryptographic implementations, such as the NSA's Speck lightweight block cipher [88].

Laser beam is another type of optical attacks. The injected fault is similar to the one used with a camera flash, except that it is a lot more precise and is capable of always inducing faults. The main downside of this method is that it requires a high expertise. Dutertre et al. [89] explain the theory behind this technique at the lowest level.

Figure 2.10: Example of a laser fault injection station (by Riscure Laser Station 2 [19])

Figure 2.10 shows an example of a laser fault injection station made by Riscure. It contains powerful red and NIR diode lasers (respectively 14 W, and 20 W). The red laser is designed for frontside testing of smart card chips, and in combination with the optics it produces a spot size of 6 * 1.40 µm on the chip surface. The near-infrared laser is designed for backside testing of smart card chips. This powerful diode laser penetrates the chip substrate to reach the transistors. This station automates the surface scanning process, offers precise control of laser power, and

injects pulses with a small spot size. It has a precise and fast response thanks to a trigger and the ability to perform multi-glitching.

Using a laser beam, a single bit [90] in a memory can be set (from logical 0 to logical 1) or reset (from logical 1 to logical 0) by attacking either the frontside or the backside of the chip. Today, the capabilities of laser injection mechanisms make it possible to carry out attacks with multiple faults. Colombier et al. [91] use a four-spot laser bench to inject up to 4 non-contiguous bits in a single cycle, or multiple non-contiguous bits over multiple cycles. This fault injection mechanism therefore makes it possible to construct much more complex attacks, potentially capable of bypassing many countermeasures.

Breier et al. [92] studied the fault mechanism of circuit logic elements in FPGA environment, and performed a practical laser fault injection into a single bit CED-protected block cipher in Xilinx Virtex-5 FPGA. Figure 2.11 shows their setup to inject fault into a Xilinx Virtex-5 FPGA. The chip is preprocessed by a mechanical solution in order to reduce the substrate thickness to approximatively 100 µm. Thinner substrate leads to easier laser penetration, at the risk of destroying logic resources or routing channels on the chip. The laser used is a 20 W diode pulse laser with 5 times magnification lens, which reduce the effective maximum power to 10 W. The wavelength is 1064 nm and the spot size of the laser beam is approximatively 840 μm^2.

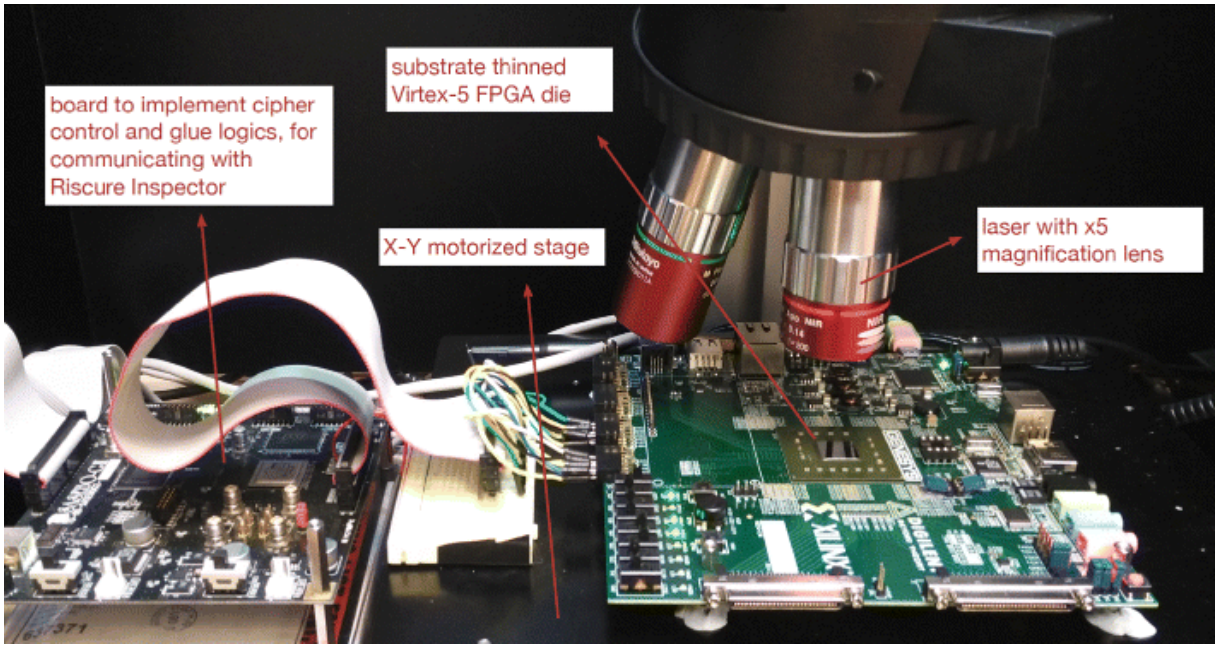


Figure 2.11: Example of a laser fault injection setup (by [92])

Focused ion beam is the most accurate and powerful fault injection technique used. Focused ion beam (FIB) enables an attacker to arbitrarily modify the structure of a circuit, reconstruct

missing buses, cut existing wires and rebuild them. FIB systems typically use liquid metal ion sources, where their low atomic mass and relatively low energy of these ions make them suitable for high-resolution imaging and precision milling of materials at the nanoscale [93].

FIB can operate at a precision of 2.50 nm, which is the size of a transistor in an actual IC. FIB workstations require very expensive consumables and a strong technical background to fully exploit their capabilities. The only limit to the FIB technology is the diameter of the atoms whose ions are used as a scalpel. Currently, the most common choice is Gallium, which sets the lower bound to roughly 0.14 nm.

These attacks are out of the scope for classical considered attackers due to the cost of the equipment needed. However, these attacks can be considered for critical systems such as military equipment. The granularity of the faults that can be introduced with FIB makes it possible to emulate both physical defects (such as stuck-at faults) and more complex logical faults.

Figure 2.12 shows the principle of how FIB works. The gallium (Ga^+) primary ion beam hits the sample surface and sputters a small amount of material, which leaves the surface as either secondary ions (i^+ or i^-) or neutral atoms (n^0). The primary beam also produces secondary electrons (e^-). As the primary beam strikes on the sample surface, the signal from the sputtered ions or secondary electrons is collected to form an image.

Torrance and James [94] report a successful reconstruction of an entire read bus of a memory containing a cryptographic key without damaging the contents of the memory.

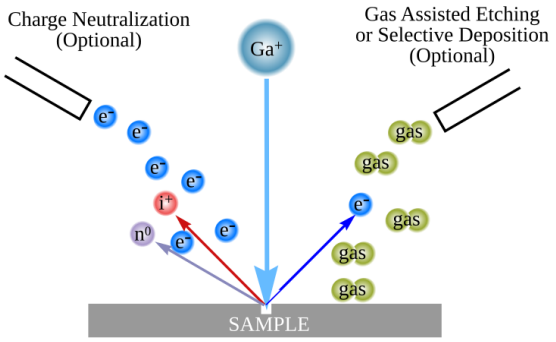


Figure 2.12: The principle of FIB (by [95])

2.3.3.2 Non-invasive attacks

Non-invasive attacks involve inducing errors in a system without physically tampering with the device. These attacks exploit external influences like electromagnetic interference, voltage glitches, or clock signal manipulation to cause faults during the device's operation. Unlike invasive methods, which require dismantling or altering the hardware, non-invasive techniques leave no physical traces, making them harder to detect. By injecting faults at precise moments,

attackers can bypass security mechanisms, retrieve sensitive data, or alter the device's intended functionality.

X-Rays is another approach to inject fault very precisely, but this method is not invasive as X-Rays can go through the IC package without the need of decapsulating it. Another advantage is that X-Ray have a lot smaller wavelength, down to 0.01 nm, than laser injection which are limited to the wavelength of their light source, down to 1 µm. The injected fault is semi-permanent, and to make it disappear, the attacker has to heat up the device. This differs from other techniques, where the fault can disappear a few cycles after injection. This technique can be compared as a non-invasive FIB techniques. X-ray provides many opportunities for attacking electronic circuits. Among them, we can note the possibility to cause permanent faults in cryptographic algorithms, deactivation of countermeasures, reprogramming of memories, etc.

Anceau et al. [96, 97] propose an approach for modifying the behaviour of a transistor in the memory of a circuit using focused X-ray beams. They use the European Synchrotron Radiation Facility (ESRF), in Grenoble, France. Grandamme et al. [98] show efficiency of X-Ray faults injection on flash and EEPROM memories for powered off devices. They also describe a fault model according to their experimental results and propose a solution to correct a part of the fault.

Clock glitches are a type of fault injection attack that targets the timing of a system's clock signal to introduce errors into its operation. It is primarily used to disrupt the normal execution of a digital circuit, such as a microcontroller or a cryptographic processor, by momentarily altering its clock frequency.

In this attack, the adversary deliberately introduces short pulses or glitches into the clock signal. These glitches can cause the system to either skip instructions, execute them incorrectly, or process data in unintended ways. By carefully timing these glitches, the attacker may manipulate sensitive operations, such as cryptographic computations, potentially exposing vulnerabilities like secret keys, bypassing security checks, or triggering unintended behaviour in the device.

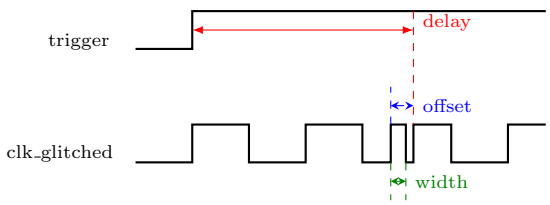


Figure 2.13: Representation of the parameters of a clock glitch attack

Figure 2.13 represent the three parameters that are taking into account for this kind of

attacks:

- Delay: the time between the rising edge of the trigger signal and the rising edge of the targeted device's clock cycle.
- Offset: determines when the glitch is applied relative to the system's clock cycle.
- Width: the duration of the glitch.

The duration of both offset and width can not be too large or too short. Because too short values will lead to too short range to obtain a timing violation, and too large values will not modify the instruction behaviour.

Figure 2.14 represent an example of a clock glitch attack, where you can see the *Normal Clock* is not faulted, and its behaviour is very regular. While, the *Glitched Clock* suffers from a glitch where an abnormal cycle is introduced and its induce an additional instruction execution. Under real conditions, the injected clock cycle would not last long enough for the instructions to execute normally. Hence, in these conditions, an instruction skip would happen.

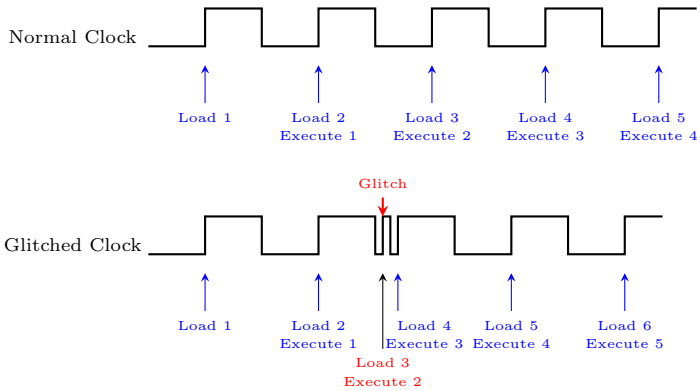


Figure 2.14: Representation of a clock glitch attack (inspired by [99])

Balasch et al. [100] show clock glitches can cause an instruction skip during the execution of a program.

Voltage glitches exploit the power supply of a digital system to introduce errors in its operation. Instead of manipulating the clock signal, this technique involves deliberately varying the voltage supplied to the system, typically by creating sharp, transient drops or spikes in the power supply (i.e. under- or overvolting) [101], or redirecting it to ground to generate voltage drops, known as "glitches" in order to generate faults of one or multiple bits. This can corrupt the contents of memory units or force microprocessors to misinterpret or even skip program instructions. Such as clock glitches, voltage glitches can be used to bypass authentication mechanisms, extract cryptographic keys, or cause logic errors that undermine the security of a device.

It's a widely recognised threat in hardware security, especially in applications where physical access to devices is possible, such as smart cards, IoT devices, and hardware security modules.

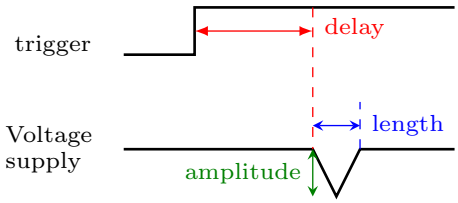


Figure 2.15: Representation of a voltage glitch attack

Figure 2.15 represent the three parameters that are taking into account for this kind of attacks:

- Delay: the time between the rising edge of a trigger signal and the injection.
- Amplitude: the voltage value of the injection or the drop introduced. In Figure 2.15 a drop in the voltage is represented, but the spike could be in the positive axis and then introduce an overvoltage in the circuit.
- Length: the time duration of the applied power variation.

Timmers and Mune [102] demonstrated voltage FIAs for Linux-based privilege escalation on an undisclosed ARM Cortex-A9-based SoC. The authors targeted the open syscall when an unprivileged application attempted to access physical memory. The application was instrumented to trigger the fault during the kernel's access control check, which caused it to be skipped. Timmers et al. [26] show the use of glitch injections on the power supply to change the CPU PC register to a predetermined address while executing random kernel syscalls, generating system crashes.

Heating attacks involve deliberately raising the temperature of a digital system or its components to induce malfunctions and errors. This type of attack exploits the fact that many electronic devices and integrated circuits are sensitive to temperature variations and may not operate reliably when subjected to abnormal thermal conditions.

On the other hand, these attacks have limitations in terms of both temporal and spatial precision. In other words, heating or cooling a device takes a long time due to thermal inertia compared to the speed of the device's calculation and hence precise attack can not be executed.

Anagnostopoulos et al. [103] present a study of data remanence effects on SRAM memories devices for temperature ranging between $-110\,°C$ and $-40\,°C$. From their results, they assess potential countermeasures against a new attack defined from data remanence.

Hutter et al. [72] heat up a microcontroller beyond operating temperature and manage to attack an RSA software implementation.

Electromagnetic fault injections (EMFI) disrupt the normal operation of a system. In this attack, an attacker generates short bursts of strong electromagnetic fields aimed at a specific part of the device, such as a microcontroller or a processor, in order to induce faults in its execution.

The goal of EMFI is to cause unintended behaviour in the target system by disturbing its internal electrical circuits. These disruptions can lead to various faults, such as skipping instructions, corrupting data, triggering incorrect logic states, or bypassing security checks. By carefully controlling the timing, location, and intensity of the EM pulses, the attacker can influence critical operations within the device, potentially gaining access to sensitive information or compromising the system's security. EMFI is particularly effective because it does not require direct physical contact with the system. The state-of-the-art EMFI setups provide millimetre-level precision in spatial location and nanosecond-level precision in the temporal location of the EM pulse. It's worth noting that EMFI can also be considered invasive. Some classify EMFI into a third category, known as semi-invasive attacks, because the package can be removed to allow direct access to the IC, improving EMFI efficiency and accuracy.

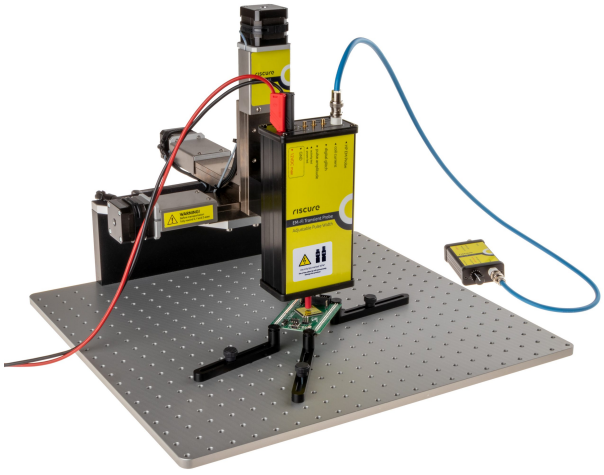


Figure 2.16: Example of an EMFI attack setup (by [104])

Dehbaoui et al. [105] succeeded in recovering the encryption key of an AES software implementation by injecting a short EM pulse on a 32-bit microcontroller. Schmidt et al. [86] use a simple gas lighter to induce EM pulses onto an 8-bit microcontroller with low spatial and temporal precision. Trouchkine et al. [24] present an approach to recover an AES key, using EMFI, by targeting the cache hierarchy and the MMU.

2.3.3.3 Fault Injection techniques summary

Table 2.2 shows a summary of all presented techniques to realise a Fault Injection Attack. Depending on the budget available for the attacker, and the required need for spatial and timing accuracy, the technique can be different.

Clock glitches, voltage glitches, heating attacks and camera flash can cost from few tens of Euros / US Dollars (USD) to less than $3,000. For EMFI attacks, Chip Shouter [21] cost around $3,000 and more precise setup can cost $30,000 [83]. These techniques are accurate and require a low to moderate expertise on the equipment and techniques. The level of expertise required depends on both the equipment and the accuracy of the attack. The more precise the equipment, the higher the level of expertise needed. On the other hand, for even more precise techniques, such as laser, FIB, or even X-Ray, the cost can go up to millions of USD/Euros as the equipment can be a lot more expensive, such as the equipment needed for X-Ray injection, but an attacker can recover a lot of secret data thanks to these attacks.

Table 2.2: Fault Injection methods summary

Technique	Precision (time)	Space accuracy	Cost	Expertise	Damage risk
Clock Glitches	High	Low	Low	Low	Very low
Voltage Glitches	Moderate	Low	Low	Low	Very low
Heating attacks	Very low	Very low	Low	Very low	Moderate
Camera flash	Moderate	Low	Moderate	Moderate	High
EMFI	High	High	Moderate	Low/Moderate	Low
Laser	Very high	Very high	High	High	Very high
Focused Ion Beam	Very high	Very high	Very high	Very high	Very high
X-Ray	Very high	Very high	Highest	Very high	Very low

2.3.3.4 Fault models

In the context of physical attacks, a fault model is a conceptual representation of how faults can occur and the effects they can have on the operation of a system. In simple terms, it describes the various ways in which a system can be altered when subjected to external perturbations. We present the most popular fault models, which are used in the literature.

Multiples studies [16, 81, 83, 106, 107] present a small overview on different fault model for Fault Injection Attacks. Different possibilities exist depending on the equipment and the effect targeted. Otto [108] presented a deep study and definitions of fault models.

With a low-cost equipment, an attacker can achieve instruction skip, random byte attacks, execution faults. While with higher cost equipment, this attacker is able to create bit-flip into the architecture, bit set/reset, or stuck-at-fault, temporal bit-flip, or spatial bit-flip.

Bit-flip [90] is the modification of a bit to the logical opposite ($0 \Rightarrow 1$ or $1 \Rightarrow 0$). Multiple bit-flips [91] are also in this category, as long as all the target bits are selected by the attacker. There is also, spatial bit-flips change the value of two bits in one or two registers at the same clock cycle. And finally, temporal bit-flips that change the value of two bits in one or two registers at two clock cycles. Bit set/reset [109] is the modification of the bit value either to logical `1` (set) or logical `0` (reset). Again, this bit can be precisely targeted by the attacker. Random byte [110] is less accurate than the previous ones as the attacker target a byte and set it to another random value (for example, in binary, from `0b01010101` to `0b00111001`). Instruction skip [111] ignores the execution of the current processed instruction. Stuck-at faults [112] permanently set the targeted data to another value.

2.4 Countermeasures against FIA

In the previous section, we showed the need to protect against fault injection attacks. In this section, we will only present the countermeasures to protect a system against fault injection attacks. Countermeasures can be implemented in software, in hardware, or even in the physical layer [14].

The objectives when implementing countermeasures are:

- to detect faults and react in accordance with a security policy (for example, tolerate them or attempt to correct them);
- to ensure that incorrect results are not usable by the attacker.

2.4.1 Countermeasures in the physical layer

Countermeasures can be implemented in the physical layer, such as sensors that detect a perturbation. He et al. [113] propose a full-digital detection logic against laser fault injection. El-Baze et al. [114] present and validates a new sensor allowing to detect EMFI. Muttaki et al. [115] introduce a universal Fault-to-Time Converter sensor that can effectively detect FIA (clock glitch, voltage glitch, laser, EMFI) while requiring minimal overhead.

2.4.2 Software countermeasures

Software countermeasures target vulnerable parts of the code (loops, memory access, etc.). They are often relatively easy to implement compared with hardware countermeasures. However, they are more likely to be bypassed, as their implementation does not take into account the system's microarchitecture. In addition, the cost to the performance of the system is significant in terms of memory requirements and execution time [14]. The principle of duplication, for example, doubles both the memory space required and the execution time for the protected sections.

A classic technique is to use temporal or spatial redundancy. Barenghi et al. [116] suggest tripling instructions by storing the results in different registers. If these registers are different, it means a fault occurred. Theißing et al. [117] implemented and systematically analysed a comprehensive set of 19 different software countermeasure strategies for protection effectiveness, time, and memory efficiency. Chamelot et al. [118] present SCI-FI, a countermeasure for Control Signal, Code, and Control-Flow Integrity against Fault Injection attacks. Laurent et al. [119] analyse some existing countermeasures and show how they handle precise faults extracted from the processor. Some countermeasures propose solutions to protect the data linked to the control flow. For exemple, Schilling et al. [120] propose protecting the calculation of conditional branches while preserving the error-detection capabilities at every stage of a conditional branch. They demonstrate this by implementing an encoded comparison using AN-codes. They also integrated this countermeasure in the LLVM compiler to automatically protect conditional branches.

However, even if those countermeasures are good against FIA, they are still sensitive against some attacks and can be bypass by analysing the processor microarchitecture. Laurent et al. [23] present an attack where they target hidden registers into a RISC-V processor. They show that even if a code is protected against FIA, they can find some vulnerabilities and bypass the software countermeasures. It is then better to directly implement hardware countermeasures at the lower level to have the best protection available.

2.4.3 Hardware countermeasures

Hardware countermeasures [14, 121] consist of adding hardware mechanisms to the system architecture, which makes them more effective. Adding a countermeasure introduces a loss of performance into the target system. Its implementation usually involves increasing the size of the hardware's area, reducing the maximum frequency, or increasing the power consumption. However, once the implementation is done, an evaluation of the protection is usually done to compare it and give some indications in terms of area, performance, or efficiency. In the state of the art, multiple solutions exist to protect a system against FIA such as information redundancy, spatial or hardware redundancy, temporal redundancy, and obfuscation.

2.4.3.1 Hardware redundancy

Hardware redundancy [122–124] countermeasure consists of duplicating the protected circuit or part of it to compare the result obtained after computation to check if there is a difference. Figure 2.17 represents the spatial redundancy. An input is sent to two or more modules (i.e. computation blocks) and the output results will be compared, to check if an error occurred. This type of countermeasure is the most direct and simplest, but at the same time, it is the one with the highest resource cost. One of the most common techniques used to implement hardware redundancy is Triple Modular Redundancy (TMR). TMR involves tripling the logic and using

voters to correct the error based on the majority. This means that in order to produce the correct output, two out of three signals must function correctly. However, there are large penalties in terms of area and power consumption with this method.

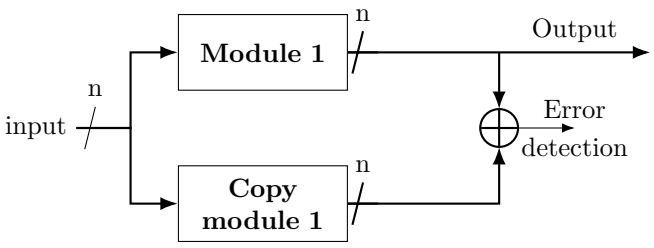


Figure 2.17: Representation of hardware spatial redundancy

2.4.3.2 Temporal redundancy

Temporal redundancy [125–127] is based on repeating operations in reverse. In this way, it is possible to check the result of an operation with its previous value. Although it has a low resource cost, it significantly increases the time required. This is because it takes twice as long to perform reverse verification operations. Furthermore, protection can be achieved with more or less resources, depending on security and redundancy levels. Figure 2.18 show how the input is saved into a register, then the value is sent to a computation module to be outputted and reversed in a reverse computation module to compare the value from the saved valued in the register. If the register's value differs from the value computed by the reverse module, it means that an error occurred, and then an error signal is emitted.

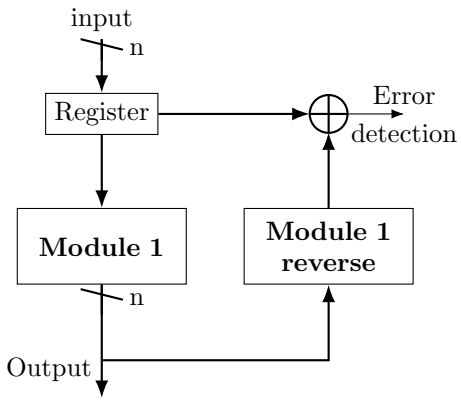


Figure 2.18: Representation of hardware temporal redundancy

2.4.3.3 Instruction replay

Another type of redundancy is to execute multiple times the same instruction or block of instructions. This redundancy, called instruction replay or instruction duplication/triplication, can be executed on one or more instructions and can be decided in software or in hardware. This solution has many advantages in terms of efficiency, but it induces large overhead in terms of performance, and area. Manssour et al. [128] present a solution to avoid large performance overhead by using dedicated instructions on a RISC-V processor. While using a very small processor, 2 stages, they have a 30% increase of area and 10% of frequency decrease. The hardware replay allows reducing the execution time and code size compared to a full software protection (for execution time, from a factor of 3 to a factor or 2, and, for code size, from a factor of 2 to a factor or 1.3).

2.4.3.4 Information redundancy

Another approach of security is the redundancy of the information. This means that additional information is added to the data to enable error detection or correction. The most important techniques in this area are Error Detection Codes (EDC) and Error Correcting Codes (ECC).

EDC [129–131] is a class of countermeasures that computes the parity (odd or even) of the protected target (e.g. registers). EDC, such as parity bits, checksums, or cyclic redundancy checks (CRC), can detect these manipulations by checking the integrity of the data or computations against redundant bits or codes. The main advantage of these countermeasures is that they inevitably detect single-bit faults with a very small overhead, unlike other previous methods. This method can only detect an error, but is unable of correcting it. This method will be further developed in Chapter 5.3 with an implementation of a simple parity code.

Error Correcting Code (ECC) [132–134], or sometimes referred to as Error Detection And Correction Code (EDAC), ensures that even if faults are injected, the system can recover the original data or identify the presence of an error by encoding the original data with additional bits (e.g. redundancy bits). This makes ECC a robust defence mechanism against fault injection attacks, improving both data integrity and system reliability. ECC can be divided into two main families and a hybrid family: Linear Block Codes, Convolutional Codes and the hybrid Turbo or Concatenated codes. Some examples of such codes are Hamming Codes, Single Error Correction Double Error Detection (SECDED), Reed-Solomon, Low-Density Parity-Check (LDPC), BCH code, and Cyclic Redundancy Check (CRC). CRC can be considered as EDC as well as ECC. This method will be developed in Chapter 5.4 with the implementation and detailed presentation of Hamming Code.

2.4.3.5 Obfuscation

Obfuscation is a technique that includes the addition of dummy cycles, during which the processor performs operations that are irrelevant to the current calculation. Another strategy is to shuffle the data to make it more difficult for the attacker to determine where to inject faults. Their effectiveness depends on their random nature. If the obfuscation is based on a constant, the attacker will only have to identify this constant to bypass the protection. On the other hand, if the obfuscation is random, the attacker will have to repeat the identification process for each new attack.

2.5 Summary

This chapter has provided an overview of the three main areas of my PhD thesis work: information flow tracking, physical attacks and countermeasures against fault injection.

The security mechanism, DIFT, is used to protect a system against software attacks such as buffer overflow, SQL injection and malware. In the remainder of this work, we are using a DIFT mechanism integrated into the hardware processor (hardware in-core DIFT) on a RISC-V processor, enabling access to the core's HDL code.

The physical attacks are diverse, ranging from the analysis of the sounds of a system or the analysis of its power consumption to fault injection using a laser or even X-rays. Their study is constantly evolving, enabling vulnerabilities in today's embedded systems to be identified with increasingly limited resources. This increases the number of potential attackers, making it all the more necessary to incorporate the concept of integrated security at the design stage, with the addition of robust countermeasures.

Finally, we provide an overview of the various existing software, hardware, and physical sensor countermeasures against fault injection attacks. These countermeasures must be integrated within certain constraints, such as effectiveness, area overhead or performance decrease.

Chapter 3

D-RI5CY — Vulnerability Assessment

Contents

3.1 Introduction

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.

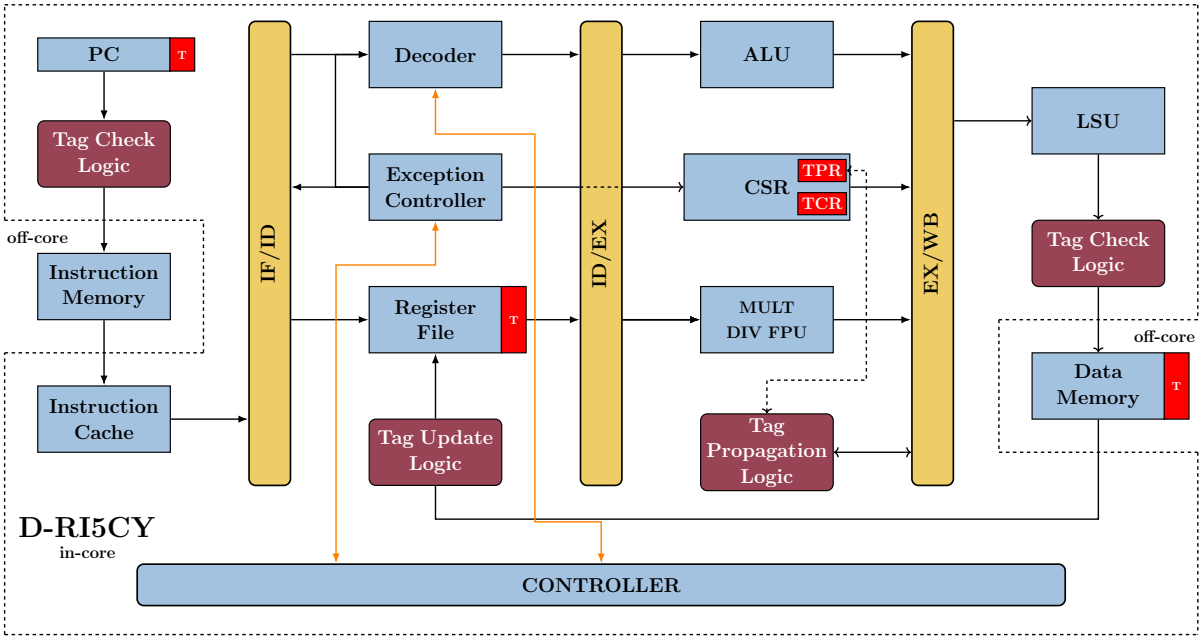


Figure 3.1: D-RI5CY processor architecture overview. DIFT-related modules are highlighted in red. (inspired by [54])

3.2 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 a open-source RISC-V core, meaning that we have the ability to access and modify the design according to our needs.

3.2.1 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 including companies such as Google, AMD, or Intel. 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.

3.2.2 DIFT design

This thesis focuses on the evaluation of a DIFT against fault injection attacks and the design of dedicated protections. 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 [54, 135] design, which utilises the RI5CY core supported by PULPino [136] and developed by PULP platform [137]. 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, ALU, and MAC operations. D-RI5CY has been developed by researchers of Columbia University, USA, in partnership with Politecnico di Torino, Italy. D-RI5CY extends the RI5CY processor to support in-core DIFT.

Figure 3.1 presents an overview of the D-RI5CY processor's architecture. DIFT modules are represented in red and dark red. 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 propagation 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 tags of the destination register *rd*,

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]*

- **p.spsb x0, offset(rt)**: sets to untrusted the security tags of the memory byte at the address of the value stored in *rt + offset*,
- **p.spsh x0, offset(rt)**: sets to untrusted the security tags 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 tags 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 (i.e. 1 tag per byte). Each data element is physically stored in memory with its associated tag. However, a tag can only have two values as in the Register File Tag, the tag is on one bit.

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 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

Table 3.2: Tag Propagation Register configuration

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

Table 3.3: Tag Check Register configuration

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

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, TPR and TCR are configured from the default security policy. Then at program startup ②, the tags are set to *trusted* (i.e, set to 0) or *untrusted* (i.e, set to 1) depending on their source or according to the code of the program as the developer can specify some untrusted part of his code. The tag propagation ③ and verification ④ happen in the D-RI5CY pipeline in parallel with the standard behaviour, without incurring any latency overhead.

3.2.3 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.

In order to specify an untrusted part in the code, the developer has to use an assembly line in C which is constructed from keywords *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`".

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 to 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.

In terms of security policy, depending on which policy is used in Table 3.2 and Table 3.3, we would 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 of 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*, implementing security policy 2 from Table 3.2 and Table 3.3. The two other use cases will be presented in the following section 3.3.

Listing 3.1: Compare/Compute C Code

```
1   int main(){
2       int a, b = 5, c;
3       register int reg asm("x9");
4       a = reg;
5       asm volatile("csrw 0x700, tprValue");
6       asm volatile("csrw 0x701, tcrValue");
7       asm volatile("p.spsw x0, 0(\%0);" :: "r" (&a));
8       c = (a > b) ? (a-b) : (a+b);
9           //42c:   ble a4,a5,448
10          //430:   addi a5,s0,-16
11          //434:   lw a4,-12(a5)
12          //438:   addi a3,s0,-16
13          //43c:   lw a5,-4(a3)
14          //440:   sub a5,a4,a5
15          //444:   j 45c
16          //448:   addi a5,s0,-16
17          //44c:   lw a4,-12(a5)
18          //450:   addi a3,s0,-16
19          //454:   lw a5,-4(a3)
20          //458:   add a5,a4,a5
21          //45c:   sw a5,-24(s0)
22      return EXIT_SUCCESS;
23  }
```

3.3 Use cases

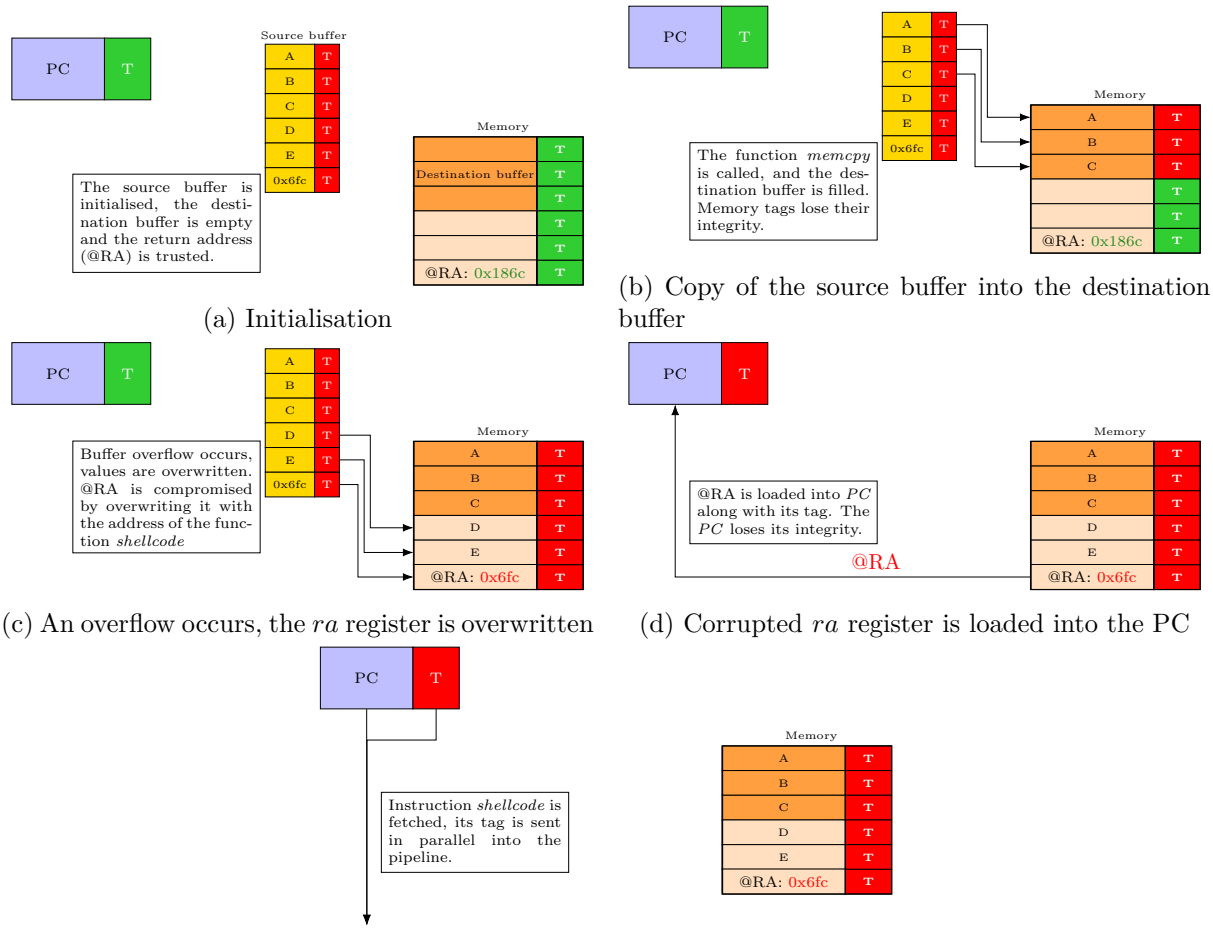
This section details the considered use cases in our work. The first two use cases come from the original paper [54]. The third use case, presented in section 3.2.3, is a home-made case which is used to stimulate DIFT elements that are not in others use cases.

3.3.1 First use case: Buffer Overflow

The first use case involves exploiting a buffer overflow, potentially leading to a Return-Oriented Programming[1] (ROP) attack[2] and the execution of a shellcode.

The attacker exploits the buffer overflow to access the return address (*RA*) register. 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 its 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. When the function returns, the corrupted *RA* register is loaded into the *PC* via a *jalr* instruction (Figure 3.2d). This hijacks the execution flow, causing the first shellcode instruction to be fetched from address: *0x6fc* (Figure 3.2e). 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. At this moment, the DIFT mechanism detects the untrusted tag and as the security policy do

1. https://en.wikipedia.org/wiki/Return-oriented_programming
2. https://github.com/sld-columbia/riscv-dift/blob/master/pulpino_apps_dift/wilander_testbed/



(a) Initialisation

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

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

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

(e) PC address instruction is fetched

Figure 3.2: Representation of how the ROP attack works

not allow executing an untrusted PC, an exception will be raised and the application will be stopped. 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.

Listing 3.2: Buffer overflow C code

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

3.3.2 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 [138] 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 is copied into another variable '`a`'. The lines 13-14 are used to initialise the tag associated to the variable '`a`'. This variable '`a`' is user-defined, so it is tagged as untrusted for

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

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. Table 3.4 represents the different steps to overwrite the memory with the exact address of the malicious function. We can see that after each write and the right shift of the writing, the address appears. Finally, we have the address '*000002000003E0*' in memory from 'A+2' to 'A-4' but as an address is on 32-bits in our architecture, the address fetched by the pipeline is only '*000003E0*'. In this use 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.

Listing 3.3: WU-FTPd C code

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

3.3.3 Summary

To summarise, these three use cases allow stimulating each element of the DIFT mechanism. Consequently, they can be used to study the impact of FIA into this mechanism. The next section studies the behaviour and assesses the DIFT against FIA.

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

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**

3.4 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 [139].

We evaluate the design by conducting fault injection campaigns. By analysing the results of these campaigns, we can determine which specific registers are vulnerable. This evaluation is performed for each individual use case previously presented, allowing for a more detailed analysis. It also helps us to understand how the error tag propagates through the system and is subsequently detected before triggering an exception.

3.4.1 Fault model for vulnerability assessment

In this vulnerability assessment, we consider an attacker able to inject faults into DIFT-related registers leading to *bit set*, *bit reset*, and *single bit-flip in one register at a given clock cycle*. As discussed in section 2.3.3.4, these fault models are the main fault models used in FIA for the most precise methods, such as laser fault injection. There is also *skip instruction* fault

model which is often used but as we do not target the configuration of the DIFT, we do not attack instructions but only registers. 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.

3.4.2 First use case: Buffer overflow

Table 3.6 shows that 24 fault injections in five 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 12 successes over 24. 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*. It is worth noting that register *memory_set_o_tag* is not in the Figure 3.3 of tag propagation but is vulnerable and create a success for bypassing the DIFT in our tests in simulation.

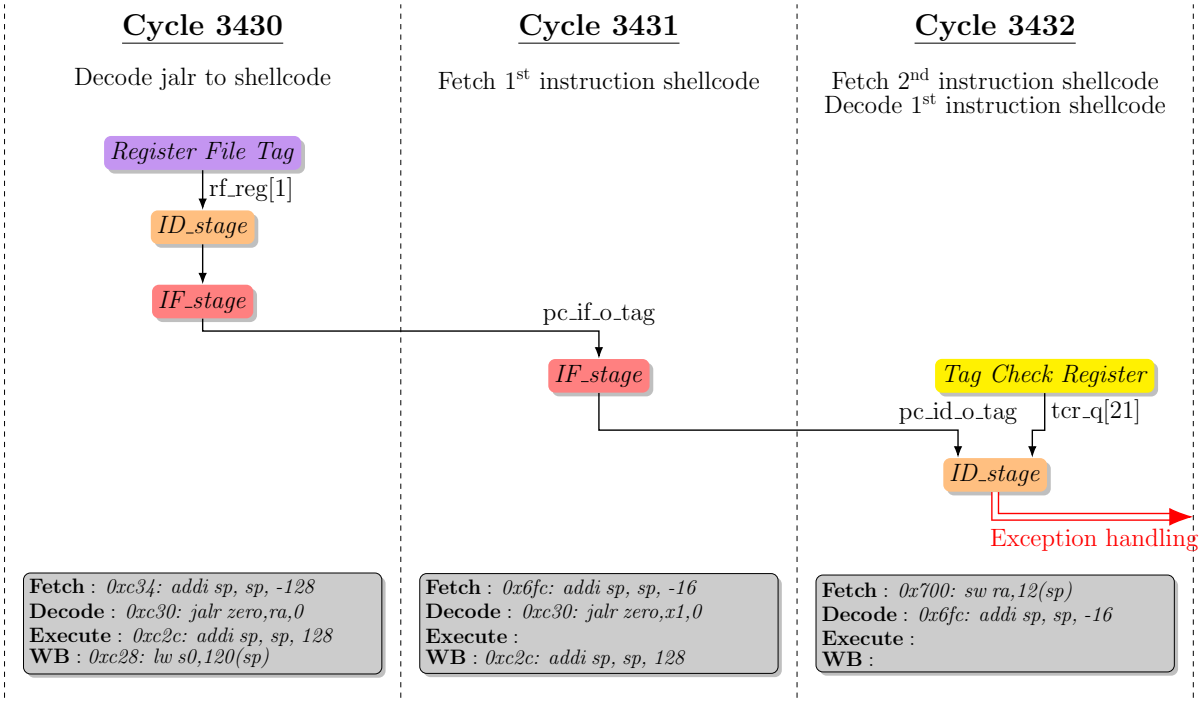


Figure 3.3: Tag propagation in a buffer overflow attack

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										✓		✓			
memory_set_o_tag		✓	✓												
rf_reg[1]							✓		✓						
tcr_q	✓			✓			✓			✓			✓		
tcr_q[21]			✓			✓			✓			✓			✓
tpr_q	✓	✓		✓	✓										
tpr_q[12]			✓			✓									
tpr_q[15]			✓			✓									

Based on these results, we can 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 to 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.

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.

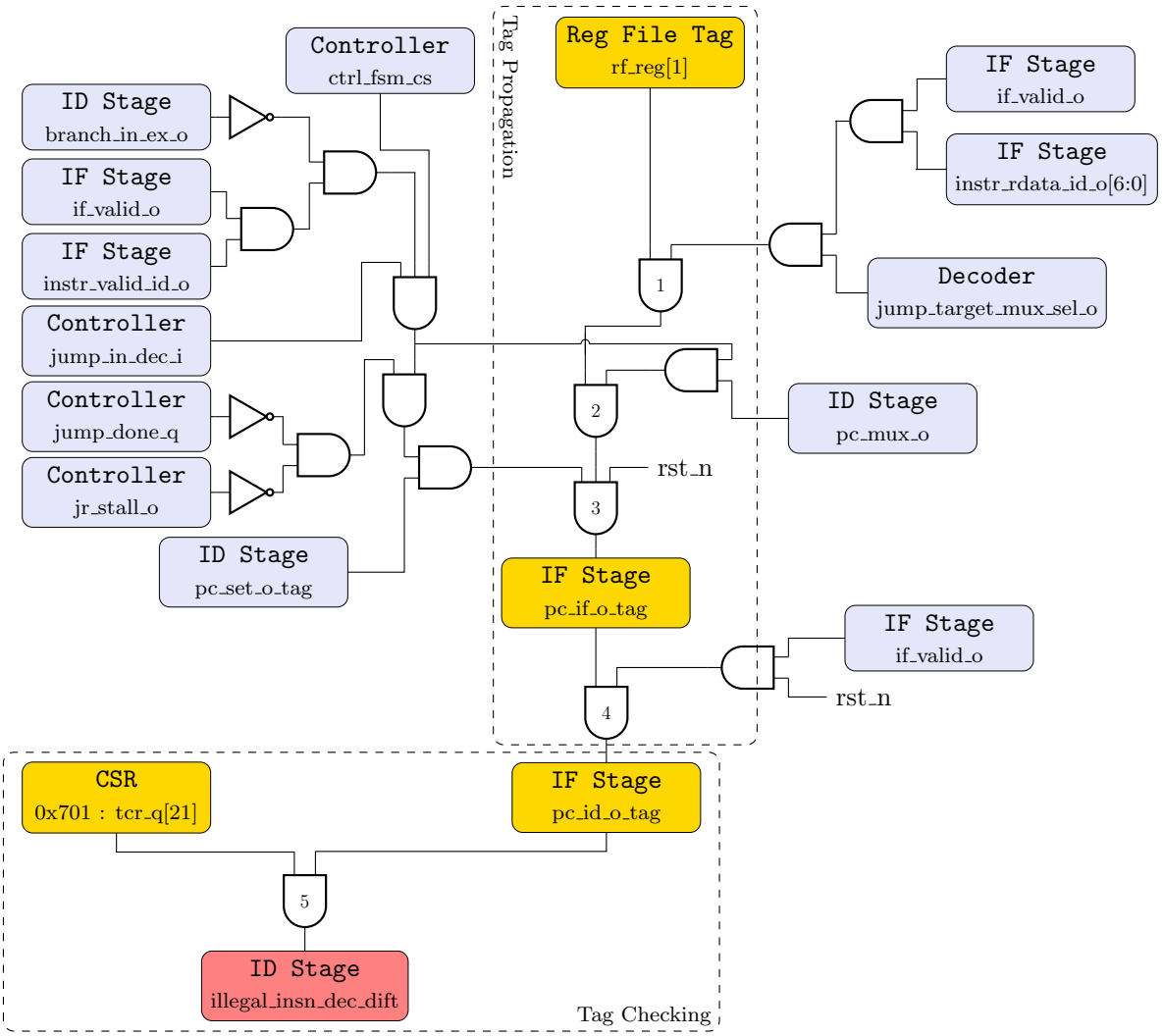


Figure 3.4: Logic description of the exception driving in a buffer overflow attack

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

Table 3.7, in page 53, 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

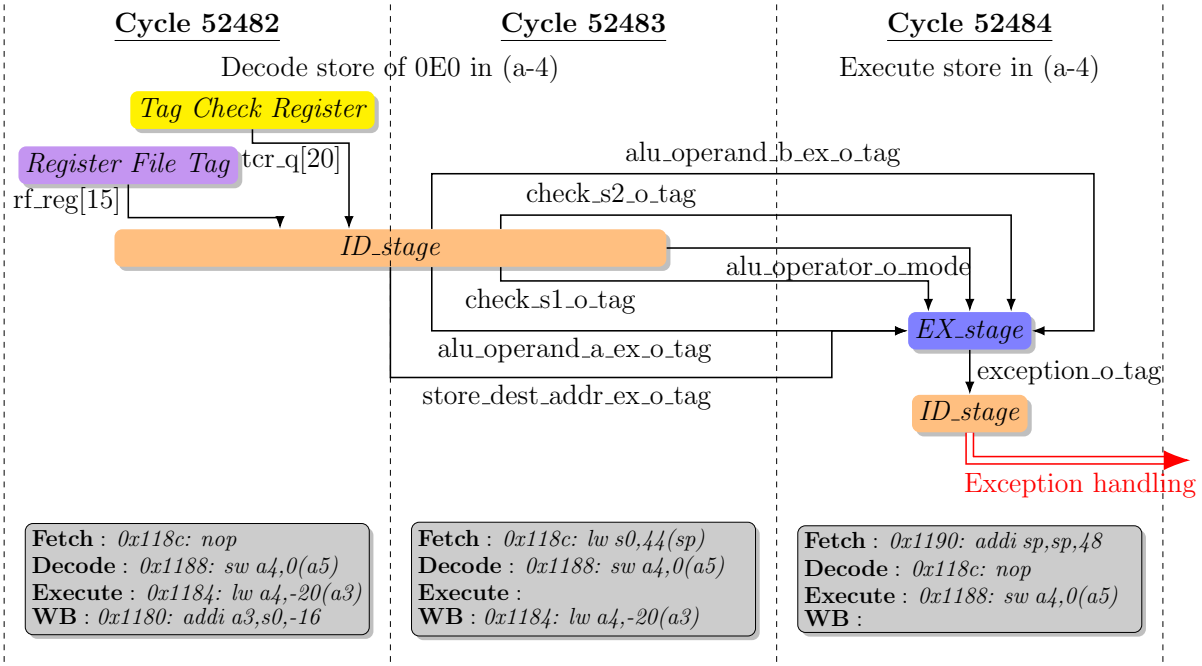


Figure 3.5: Tag propagation in a format string attack

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																						✓		✓

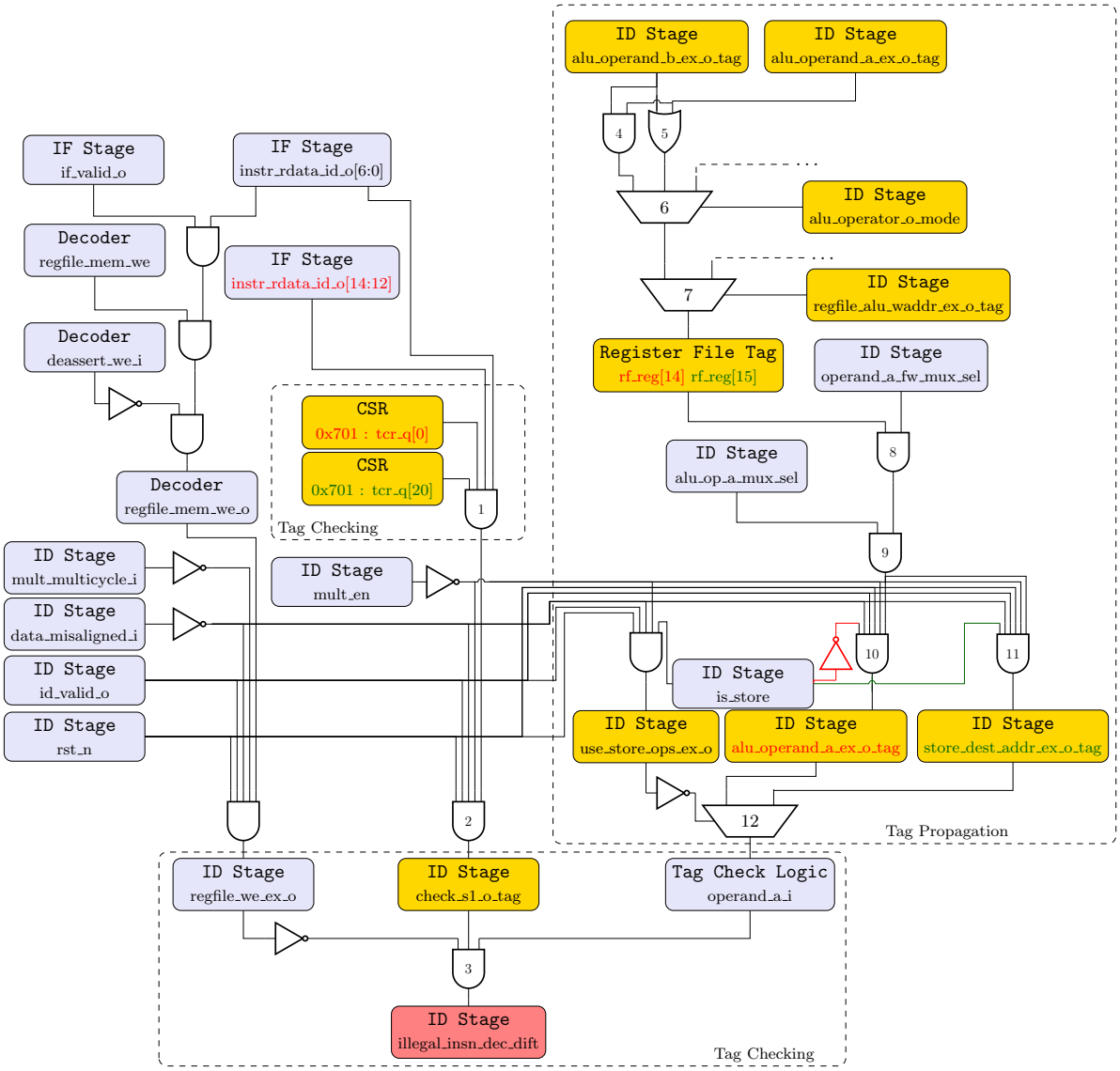


Figure 3.6: Logic description of the exception driving in a format string attack

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

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											✓	✓

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'.

3.4.4 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`

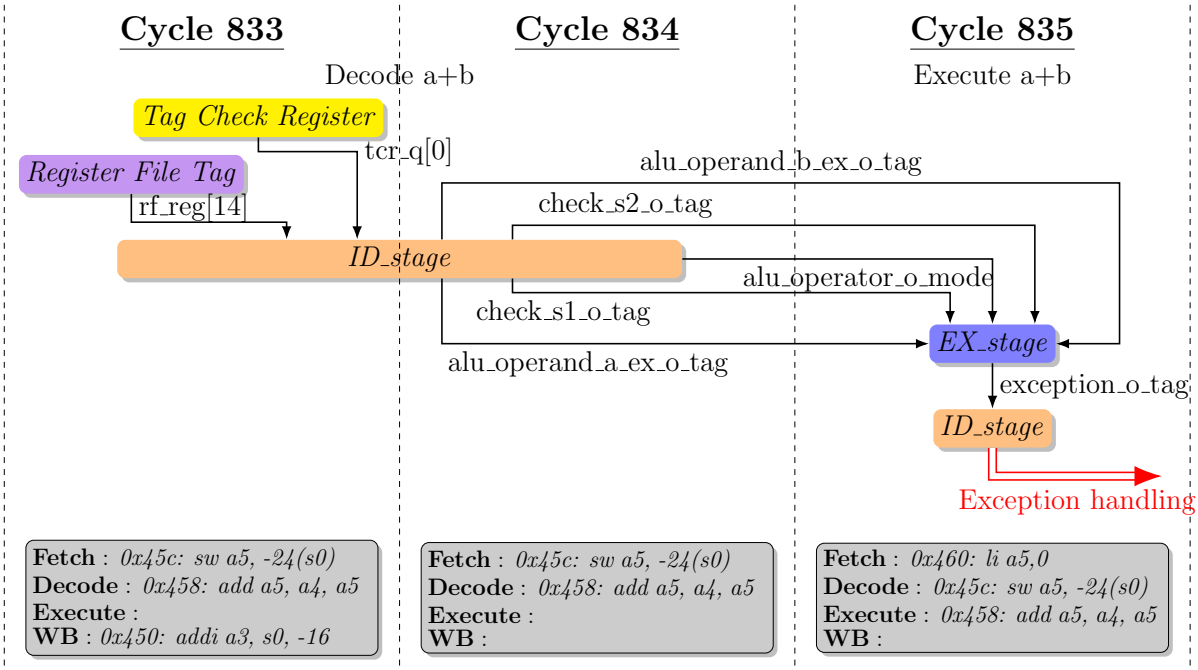


Figure 3.7: Tag propagation in a computation case with the compare/compute use case

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_*

Table 3.9: Results for *bit reset* for the baseline version

	Crash	Silent	Delay	Success	Total	Execution time
Buffer Overflow	0	320	1	9 (2.73%)	330	0:04
WU-FTPd	0	424	0	16 (3.64%)	440	0:47
Compare/Compute	0	213	0	7 (3.18%)	220	0:01

Table 3.10: Results for *bit set* for the baseline version

	Crash	Silent	Delay	Success	Total	Execution time
Buffer Overflow	0	320	7	3 (0.91%)	330	0:04
WU-FTPd	0	397	36	7 (1.59%)	440	0:48
Compare/Compute	0	213	5	2 (0.91%)	220	0:01

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.

3.5 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 have 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 critical.

Tables 3.9, 3.10, 3.11 present the results obtained from the campaign with their respective fault model. This vulnerability analysis revealed that the majority of weaknesses in this mechanism are caused by single bit-flips, with 51 successful faults out of 95. Furthermore, the registers involved in this mechanism are predominantly 1-bit registers, as they are used for the tag data path. This indicates that, to effectively safeguard the mechanism, the primary focus should be on protecting it against single bit-flip errors.

Table 3.11: Results for a *single bit-flip* for the baseline version

	Crash	Silent	Delay	Success	Total	Execution time
Buffer Overflow	0	738	12	12 (1.57%)	762	0:11
WU-FTPd	0	946	41	29 (2.85%)	1016	01:52
Compare/Compute	0	491	7	10 (1.97%)	508	0:02

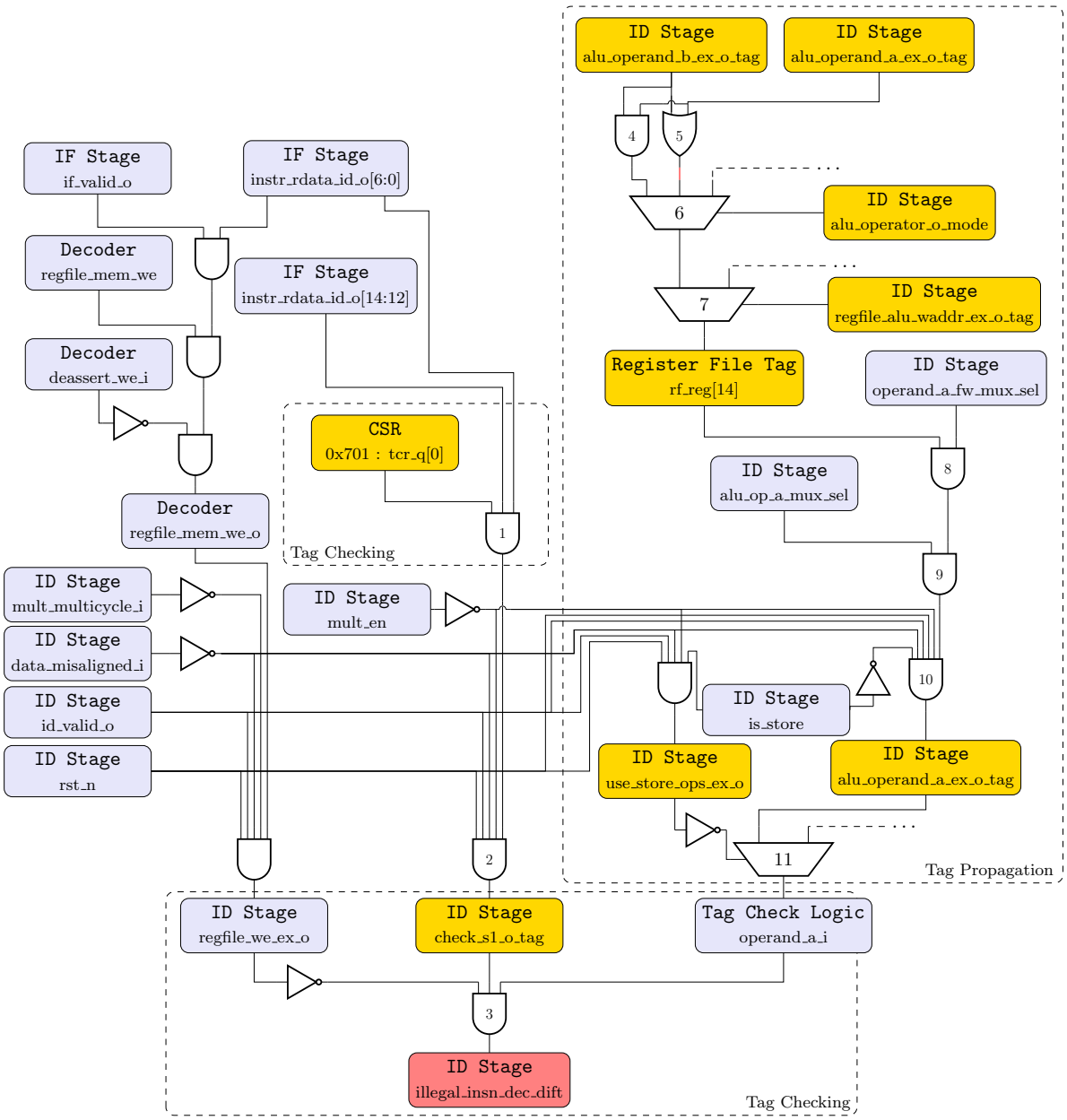


Figure 3.8: Logic representation of tag propagation in a computation case

Chapter 4

FISSA – Fault Injection Simulation for Security Assessment

Contents

4.1 Introduction

This chapter introduces and presents a tool, called FISSA – Fault Injection Simulation for Security Assessment –, created to automate fault injection attacks campaigns in simulation. This work has been published in DSD 2024 [140]. The first section presents the state of the art of existing tools for FIA campaigns in simulation, formal methods, or even perform real world attacks. The second section presents FISSA software architecture, details how FISSA works, and presents how to extend it. The third section illustrates FISSA capacity through a use case from Section 3.3. Finally, the last section discusses and draws some perspectives for the tool's development and usability.

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	[142–145]	Very low	Very high	Very low	Low	Low	Very high
Simulations	[146–155]	Very low	Very high	Low	Low/Moderate	Moderate	Low
Actual FIA	[14, 91, 98, 156–158]	Very high	Very low	Very high	Very high	Very high	Very high

4.2 Simulation tools for Fault Injection

Addressing fault injection vulnerabilities is crucial. In general, fault attacks are conducted using physical equipment. Nonetheless, another approach exists 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.

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, formal methods and simulations. Another objective of fault injection in simulation is to address safety [141]. Safety concerns revolve around unintended, accidental faults, with a focus on system reliability and resilience. The aim is to verify the system's capability to detect and recover from these faults, ensuring that no catastrophic consequences occur as a result of such failures. This process is crucial for validating the robustness of safety mechanisms in place.

Actual FIAs involve physically injecting faults into the target hardware using techniques such as variations in supply voltage or clock signal [14, 158], laser pulses [14, 91], electromagnetic emanations [14] or X-Rays [98]. 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.

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 [142] allow the analysis of a circuit design in order to detect sensitive logic or sequential hardware elements. [143], [144] and [145] 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 (e.g. if there is a loop in the design).

Many simulators for fault injection attacks exist at different levels, to achieve different ob-

jectives, such as security at gate-level, cryptographic systems, study the impact of clock glitches, or even X-Ray. They can use Artificial Intelligence (AI) to enhance the detection. Another way to simulate fault injections is to use QEMU (Quick EMUlator) [146, 147, 155]. QEMU is an open-source machine emulate the behaviour of a processor at a very fine-grain, using various optimizations to keep execution speed as close as possible to native system execution. Bekele et al [146] present a survey of QEMU-based Fault Injection techniques. After discussing the various techniques proposed in the state of the art, they classify into categories and compare them. Fault Injections simulations can be performed at processor instructions level. Authors of [148] 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. [150] introduces VerFI, a gate-level granularity fault simulator for hardware implementations. For instance, it has been used to spot an implementation mistake in ParTI [159]. 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. [149] is an open-source deterministic fault attack simulator prototype utilising the Unicorn Framework and Capstone disassembler. Tebina et al. [151] introduce Ray-Spect, a tool to simulate fault injection using parametric degradation of MOSFETs transistors, which is typical of X-ray fault injection. Wang et al. [152] developed a framework for fault injection assessment at gate-level with design specific security properties. Grycel et al. [153] present, SimpliFI, a simulation methodology to test fault attacks on embedded software using a hardware simulation of the processor running the software. It relies on post-layout netlist simulations to study the impact of fault injection techniques such as clock glitches.

In this work, we focus on RTL 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. Post-synthesis, timed, and post-implementation simulations can be more difficult to apprehend. This is because HDL synthesis alters the names of the various hardware elements, making it more difficult to find the various elements targeted in the behavioural section. Behavioural simulation-based fault injection offers the advantage of enabling designers to test their system at the early beginning of 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. Table 4.1 shows that 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".

4.3 FISSA

This section presents our open-source tool, FISSA, available on GitHub [160] under the CeCILL-B licence.

4.3.1 Main software architecture

FISSA is designed to help circuit designers to analyse, at the early beginning of the development, the sensitivity to FIA of the developed circuit. FISSA relies on behavioural simulations. Figure 4.1 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 4.3.3, 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 4.3.4, 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 [161], Verilator [162], or Vivado [163] to simulate the design according to the TCL script and generates JSON files to log each simulation.

The Analyser, detailed in Section 4.3.5, 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.

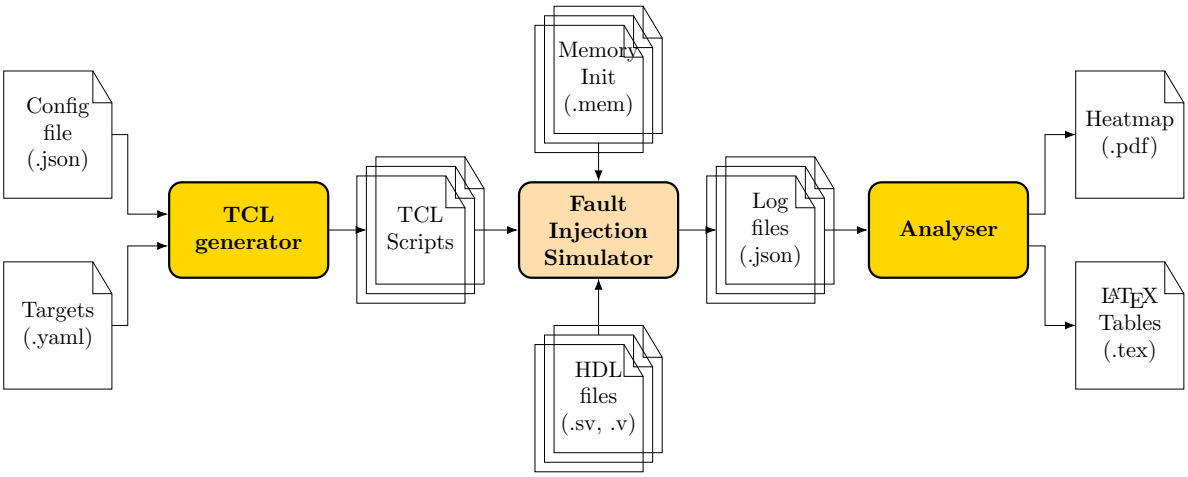


Figure 4.1: Software architecture of FISSA

Algorithm 1 shows a representation of a fault injection campaign. The algorithm requires the name(s) of the use case(s) on which the campaign will be, a set of targets (i.e. hardware elements into which a fault is to be injected), the number of bits of each target, the fault model and the injection window(s) under consideration, which identify the period(s), in a time interval between start (Δ_s) and end (Δ_e) in nanosecond, into which fault injections are carried out. The number of bits for the campaign, will be called 'κ', and 'κ_i' the number of bit of one target. The injection window will be used to calculate the number of cycle with the CPU period (Υ_{cpu}). So, the number of cycle can be determined by $nbCycles = (\Delta_e - \Delta_s)/\Upsilon_{cpu}$.

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**

4.3.2 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 target exhaustively ($nbSimulations = nbCycles * \kappa$),
- single bit-flip in two targets at a given clock cycle: we select 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 ($nbSimulations = nbCycles * C_2^{\kappa}$; with κ, the sum of the bits of each target),
- single bit-flip in two targets at two different clock cycles: we select 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 ($nbSimulations = C_2^{nbCycles} * C_2^{\kappa}$),
- exhaustive multi-bits faults in one target at a given clock cycle: we select one cycle and one target, and we try exhaustively each combination of bits (for example for a 2-bit target, it would be: 00, 01, 10, 11) and we set the target at each value ($nbSimulations = nbCycles * 2^{\kappa_i}$). 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 select one cycle and two targets, and we try exhaustively each combination of bits (for example for a 2-bit

target, it would be: 00, 01, 10, 11) for each target and we set them to each value (nbSimulations = $nbCycles * 2^{\kappa_{1i}} * 2^{\kappa_{2i}}$). The user must be vigilant about the size of their targets, as a register can be 32 bits or even up to 64 bits. Exhaustively testing each possible value for such large registers can be extremely time-consuming. For a 32-bit register, for example, the total number of simulations would reach 2^{32} (around 4 billion), which could lead to an astronomical amount of time and computational effort.

4.3.3 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 4.2 details the *TCL Generator* software architecture. 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

Listing 4.2: Example of a FISSA target file

```
---
## 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:
...
```

(in ns) of the HDL design and the maximum number of simulated 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 written. 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. This approach allows for obtaining comparative results after a fault has occurred, making it possible to determine the specific effects and consequences of the injected fault. By comparing the system's

behaviour before and after the fault injection, it becomes easier to identify what was impacted and how the fault influenced the system's operation.

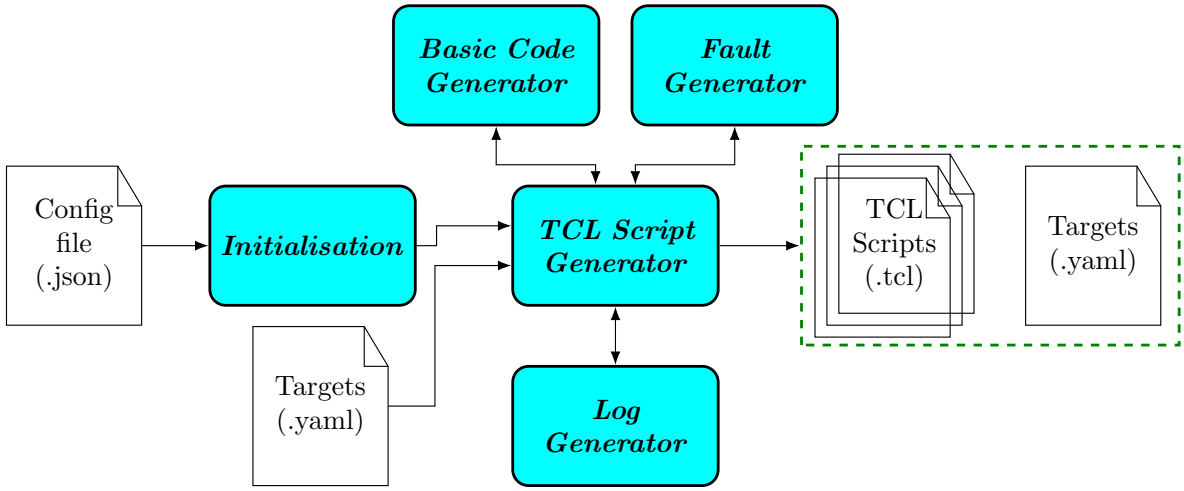


Figure 4.2: Software architecture of the TCL Generator module

Algorithm 2 depicts the pseudocode of a simulation of a fault injection, showcasing requirements, and each state with essential parameters. Additionally, the corresponding Python class from Figure 4.2 is added for each line. Line 5 in Algorithm 1 corresponds to Algorithm 2. This algorithm is executed multiple times with different inputs to build a TCL script.

Algorithm 2 FIA simulation pseudocode

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

4.3.4 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. A fault has been injected 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).

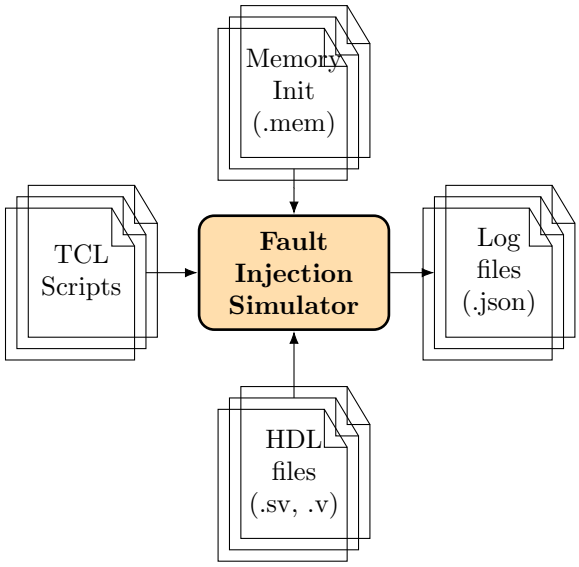


Figure 4.3: Fault Injection Simulator architecture

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.

4.3.5 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

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
}
```

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* [164] which relies on *Matplotlib* [165]. 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.

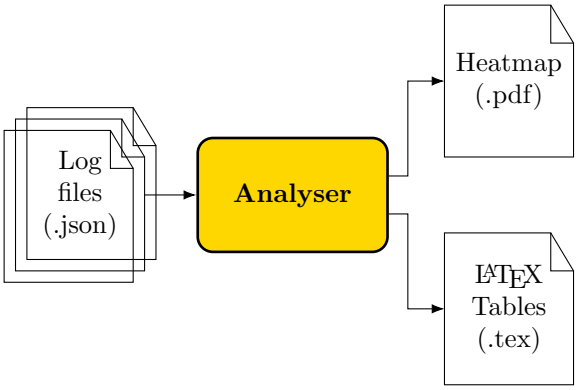


Figure 4.4: Analyser architecture

4.3.6 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. For instance, in Questasim, injecting a fault into a target can be accomplished with the command: "*force <object_name><value>-freeze -cancel <time_info>*" [166], whereas in Vivado, the equivalent command is: "*add_force <hdl_object><values>-cancel_after <time_info>*" [167]. There are some subtle differences between these two software applications that need to be taken into consideration in order to extend FISSA. These distinctions may affect the functionality or compatibility, so addressing them is crucial for a successful adaptation.

4.4 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 3.3.

4.4.1 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 4.4.2. 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 last status refers to 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

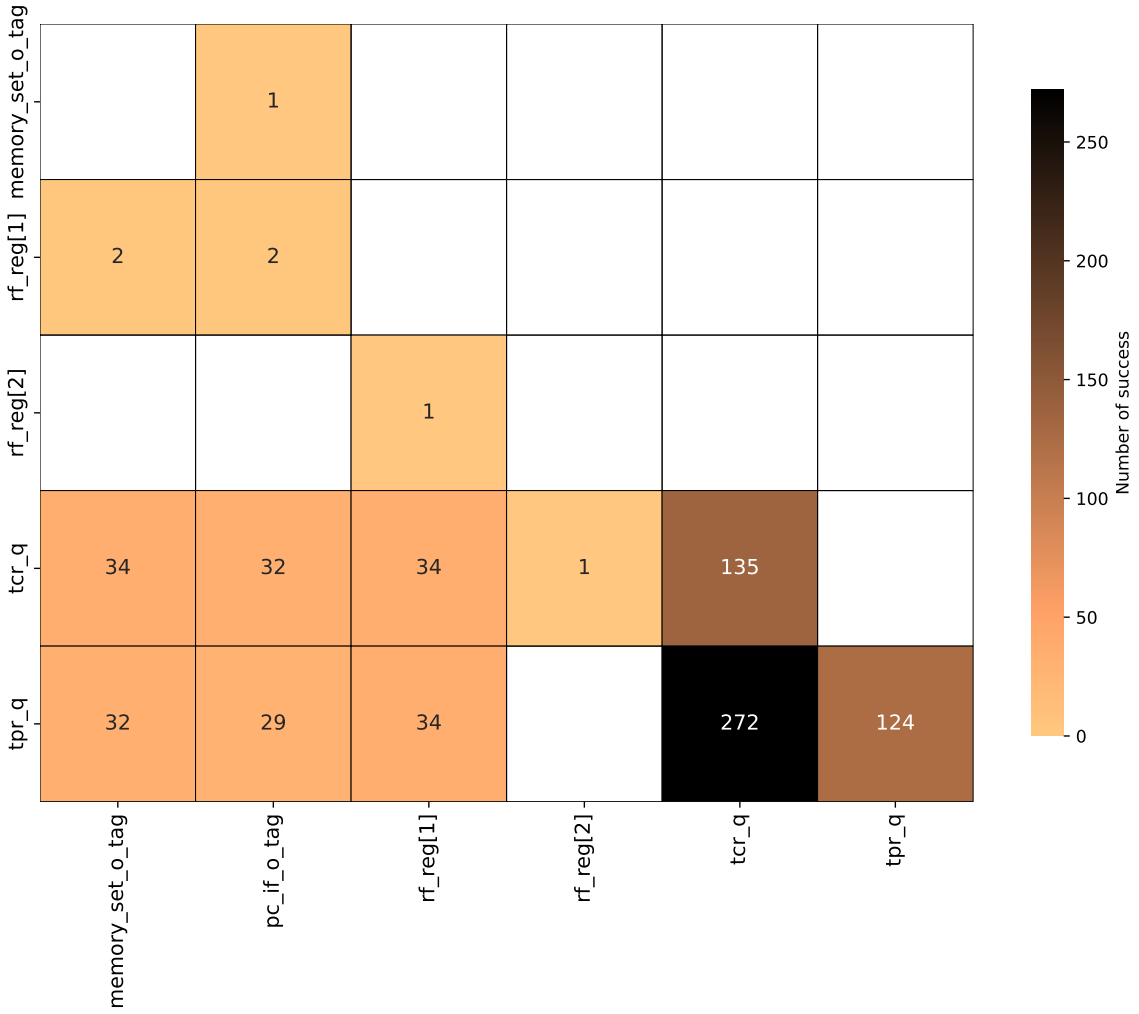


Figure 4.5: Extract of the heatmap generated according to the single bit-flip in two targets at a given clock cycle fault model

are four of the seven fault models defined in Section 4.3.2: *target set to 0*, *target set to 1*, *single bit-flip in one target at a given cycle*, and *single bit-flip in two targets at a given cycle*.

Four FIA simulation campaigns are performed to evaluate the design against the four 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.

4.4.2 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),

Table 4.2: Results of fault injection simulation campaigns

Fault model	Crash	Silent	Delay	Success	Total	Simulation time
Set to 0	0	320	1	9 (2.73%)	330	0h04
Set to 1	0	320	7	3 (0.91%)	330	0h04
Single bit-flip in one target at a given clock cycle	0	738	12	12 (1.57%)	762	0h11
Single bit-flip in two targets at a given clock cycle	0	45,097	1,503	1,406 (2.93%)	48,006	13h43

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]			✓			✓									

128 GB RAM, Ubuntu 20.04.6 LTS and Questasim 10.6e.

Table 4.2 summarises the outcomes of the four 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, which represent 0.91% of the total number of simulations, whereas the '`single bit-flip in two targets at a given clock cycle`' fault model leads to 1,406 successes, which represent 2.93% of the total number of simulations. 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 models with multiple registers faulted or with a high number of successes, where the table may become unwieldy, Figure 4.5 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 at-

tack. 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 4.5).

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 4.5 shows an extract of the heatmap generated for the *single bit-flip in two targets at a given clock cycle* fault model. For simplicity, only 5 registers are represented. The full figure will be presented in Chapter 6. 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 4.4.1. We can note that this colour scale, in our case, range from 0 to 272. This figure highlights the registers that are critical to a specific fault model, enabling the designer to evaluate the design and determine where protection is needed and at what level. It provides a clear indication of which areas require minimal protection and which ones demand a very high level of security. All of this information allow the designer to prioritise countermeasures according to allocated budget, protection requirements, etc. To give an example, it 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 3.3, 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 a ROP attack with 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 for `tcr_q` and `tpr_q` are very high priority as they lead to 37.77% success on this fault model (i.e. with all registers taken into account, see Table 4.2). In addition, we can see that a register produce a low number of successes, such as `rf_reg[2]`; this register is then not the highest priority for protection for the designer.

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 4.5 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.

4.5 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. 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.

4.6 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

Error Detection and Correction Codes to Protect an In-Core DIFT against FIA

Contents

5.1 Introduction

Previous chapters have shown that the D-RI5CY's DIFT security mechanism is vulnerable to FIAs, mainly due to single-bit flips. This D-RI5CY essentially uses single-bit registers, as it relies on 1-bit tags.

In this chapter, we present two countermeasures in order to protect the DIFT against fault injection attacks. The first countermeasure implemented to detect and prevent the use of corrupted data is simple parity. We selected the simple parity code as the error detection countermeasure because of its suitability and limited overhead. However, parity codes are limited in that they

can only detect, but not correct, single-bit errors. The second countermeasure is implemented to detect any single-bit errors that may occur, but also to correct them without time overhead. With this countermeasure, we want to correct to the nearest cycle so that the fault cannot propagate and give a potential attacker the impression that the fault he injected had no effect on the system. This chapter presents the work done during a 4-month research stay, funded by the *Collège Doctoral de Bretagne*, *GDR ISIS (CNRS)*, and the *Université Bretagne Sud*, within the *ALaRI* laboratory (*Advanced Learning and Research Institute*) in the *Università della Svizzera Italiana* in Lugano, Switzerland. This work has been published in ISVLSI 2024 [168].

The first section of this chapter presents the different fault models considered. Then, the second section presents simple parity and details its implementation. Afterwards, the third section presents Hamming code principles, with a simple example, and details our implementation. Finally, we discuss these countermeasures and compare them.

5.2 Fault models considered in this chapter

In Chapter 3, we assessed the D-RI5CY design by considering *single bit-flip in one register at a given clock cycle*, *bit reset*, and *bit set* fault models. The conclusion of this chapter was that the D-RI5CY is vulnerable mostly to single bit-flip, due to the fact that this DIFT design is mostly built around 1-bit registers for tag propagation.

In this chapter, we consider an attacker able to inject faults into DIFT-related registers, leading to single bit-flips at any position of the targeted register. To reach this objective, any DIFT-related register maintaining 1-bit tag value, driving the tag propagation or the tag update process or maintaining the security policy configuration can be targeted. Studies presented in [169, 170] have shown that such precise single bit-flip attacks targeting registers can be performed using, for example, laser shots. We also consider an attacker able to inject a *single bit-flip in two registers at two distinct clock cycles*, with a minimum delay of one clock cycle. Nowadays, more and more platform exist to perform multi-bits faults on different targets [171, 172].

5.3 Simple Parity

Parity codes represent one of the simplest and most fundamental methods for error detection in digital communication systems. Utilised across a wide range of applications, parity codes help ensure data integrity by adding a single parity bit to a block of data. This bit acts as a basic error-detection mechanism, enabling the identification of single-bit errors during transmission. Parity codes are commonly classified into two types: even parity and odd parity. In an even parity system, the parity bit is set such that the total number of 1s in the data block, including

Table 5.1: D-RI5CY Registers Details List

Register Name	Module	Size
pc_id_o_tag	Instruction Fetch Stage	1
pc_if_o_tag	Instruction Fetch Stage	1
alu_operand_a_ex_o_tag	Instruction Decode Stage	1
alu_operand_b_ex_o_tag	Instruction Decode Stage	1
alu_operand_c_ex_o_tag	Instruction Decode Stage	1
alu_operator_o_mode	Instruction Decode Stage	2
check_d_o_tag	Instruction Decode Stage	1
check_s1_o_tag	Instruction Decode Stage	1
check_s2_o_tag	Instruction Decode Stage	1
is_store_post_o_tag	Instruction Decode Stage	1
memory_set_o_tag	Instruction Decode Stage	1
regfile_alu_waddr_ex_o_tag	Instruction Decode Stage	5
register_set_o_tag	Instruction Decode Stage	1
store_dest_addr_ex_o_tag	Instruction Decode Stage	1
store_source_ex_o_tag	Instruction Decode Stage	1
use_store_ops_ex_o	Instruction Decode Stage	1
rf_reg[0]	Register File Tag	1
rf_reg[1]	Register File Tag	1
rf_reg[2]	Register File Tag	1
rf_reg[3]	Register File Tag	1
rf_reg[4]	Register File Tag	1
rf_reg[5]	Register File Tag	1
rf_reg[6]	Register File Tag	1
rf_reg[7]	Register File Tag	1
rf_reg[8]	Register File Tag	1
rf_reg[9]	Register File Tag	1
rf_reg[10]	Register File Tag	1
rf_reg[11]	Register File Tag	1
rf_reg[12]	Register File Tag	1
rf_reg[13]	Register File Tag	1
rf_reg[14]	Register File Tag	1
rf_reg[15]	Register File Tag	1
rf_reg[16]	Register File Tag	1
rf_reg[17]	Register File Tag	1
rf_reg[18]	Register File Tag	1
rf_reg[19]	Register File Tag	1
rf_reg[20]	Register File Tag	1
rf_reg[21]	Register File Tag	1
rf_reg[22]	Register File Tag	1
rf_reg[23]	Register File Tag	1
rf_reg[24]	Register File Tag	1
rf_reg[25]	Register File Tag	1
rf_reg[26]	Register File Tag	1
rf_reg[27]	Register File Tag	1
rf_reg[28]	Register File Tag	1
rf_reg[29]	Register File Tag	1
rf_reg[30]	Register File Tag	1
rf_reg[31]	Register File Tag	1
rs1_o_tag	Execute Stage	1
tcr_q	Control and Status Registers	32
tpr_q	Control and Status Registers	32
data_type_q_tag	Load/Store Unit	2
data_we_q_tag	Load/Store Unit	1
rdata_offset_q_tag	Load/Store Unit	2
rdata_q_tag	Load/Store Unit	4

the parity bit, is even. Conversely, in an odd parity system, the parity bit is adjusted so that the number of `1` is odd.

5.3.1 Simple parity in a nutshell

The key advantage of parity codes lies in their simplicity and low overhead. A single parity bit, added to each data block, is sufficient to detect any single-bit error in the block. This one bit store the parity of the initial message. Figure 5.1 shows how the data, in blue, and the parity bit, in red, are associated to form an encoded data.

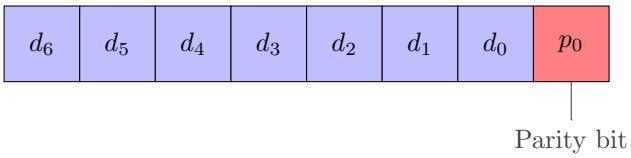
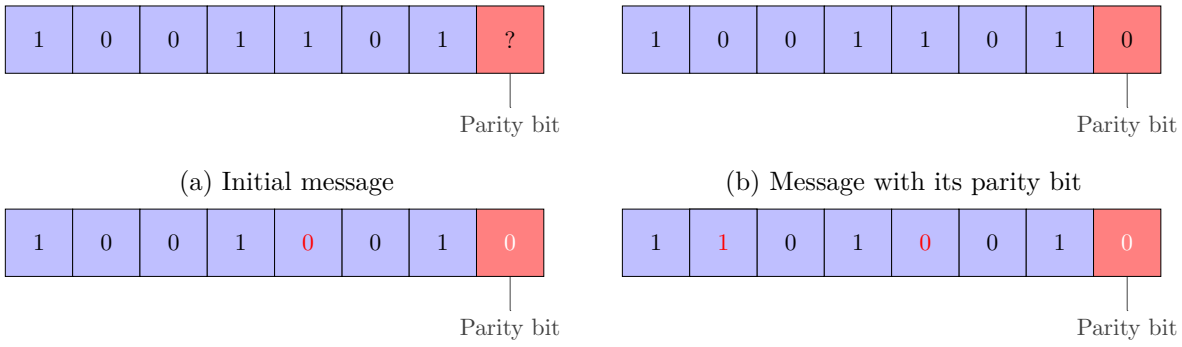


Figure 5.1: Simple Parity – functioning

Equation 5.1 shows how the parity bit is computed. Each bit of the initial message is XOR'd to calculate parity.

$$p_0 = d_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_5 \oplus d_6 \tag{5.1}$$

Figures 5.2a and 5.2b show an example of a message with its parity bit. The message is `0b1001101`. Hence, as there is an even number of '`1`', the parity bit is set to '`0`'.



(a) Initial message

(b) Message with its parity bit

(c) Single-bit fault inside the message

(d) Two single-bit faults inside the message

Figure 5.2: Example of a simple parity calculation and its fault detection capacity

Figures 5.2c and 5.2d present, respectively, two examples of when a fault occur and when two faults happen. In the first example, Figure 5.2c, the bit d_2 (from Figure 5.1), in red, is faulted. As the faulted message is `0b1001001`, it means that the new calculated parity bit value should be

Table 5.2: DIFT-related protected registers – simple parity

	Protected register	Number of bits	Number of protected bits	Number of parity bits
Group 1	TCR	32	22	1
Group 2	TPR	32	22	1
Group 3	Register File (Tag)	32	32	1
Group 4	Tag destination address	5	5	1
Group 5	16×1-bit registers 3×2-bit registers 1×4-bit register	26	26	1
Total		127	107	5

`1`. Hence, the fault will be detected as the parity bit differs from the original computed message (Figure 5.2b). In the second case, two faults happen in the message at bits d_2 and d_5 (from Figure 5.1). So, the faulted message is `0b1101001`, then, when the new parity bit is calculated, the parity bit value will not change as there is still an even number of `1` compared to the initial message. This shows the limitation of this error detection code.

5.3.2 Strategy 1: Minimisation of redundancy bits

In order to implement simple parity, we decided, in a first approach, to optimise the number of parity bits. We had many choices, but we decided to form five groups. These groups are composed of one or more register according to their criticality. All 55 registers are shown in Table 5.1. This Table Firstly, the two registers that contain the security policy, TCR and TPR, are highly critical. As a result, we have chosen to form a separate group for each of them. Although these registers are 32 bits long, only 22 bits are fully utilised in the current implementation, making bits 22 to 31 unnecessary. Therefore, we have decided not to protect these unused bits or include them in parity calculations. Secondly, the third logical group consists of keeping the 32 registers of the register file tag together. Since these registers are already grouped, it makes sense to maintain this grouping. This leaves us with one 5-bit register, sixteen 1-bit registers, three 2-bit registers, and one 4-bit register. The 5-bit register is used to store the tag destination address, which is critical. As such, we have decided to create a dedicated group for it. The remaining 20 registers, which total 26 bits, are combined into a fifth group. Table 5.2 shows the five groups formed to implement the protection for 107 bits in total. One parity bit protects each group.

Figure 5.3 presents our proposed implementation for the simple parity. This implementation is straightforward. To protect a register (shown in blue), the input is directed simultaneously to both the protected register and an encoder. The encoder calculates the parity using combinatorial logic, storing the resulting parity bit in a separate register, depicted in red in the figure. The parity bit is stored in this register during the same cycle as the input value is stored in the protected register. Subsequently, the decoder computes the parity of the protected register and

compares it with the parity bit stored in the parity bit register. If a difference is detected, it indicates the injection of a fault, which causes an alert signal to be raised.

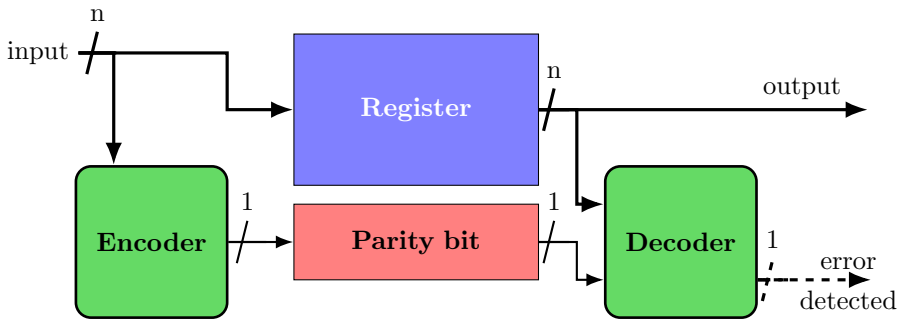


Figure 5.3: Implementation of simple parity

5.4 Hamming Codes

In digital communication and error correction theory, Hamming codes represent a pioneering development in ensuring data integrity during transmission over unreliable channels. Developed by Richard Hamming in 1950 [173], this class of error-correcting codes is designed to detect and correct single-bit errors and detect, without the correction part, two-bit errors. The Hamming code is a linear block code that enhances data transmission reliability by introducing redundancy in a structured manner.

The importance of Hamming codes lies not only in their ability to maintain the integrity of data but also in their efficiency relative to other early error correction schemes. As such, Hamming codes have found wide application in areas where high data accuracy is required, such as computer memory systems, telecommunications, and satellite communication. Despite the emergence of more sophisticated error-correcting codes in modern systems, the simplicity and effectiveness of Hamming codes make them a foundational topic in the study of error correction algorithms.

5.4.1 Hamming Code in a nutshell

The fundamental principle behind the Hamming code is the strategic insertion of r redundancy bits at specific positions within a data block of d bits, such that $2^r \geqslant d + r + 1$. These parity bits are used to perform checks on subsets of data bits, allowing the receiver to identify and, in certain cases, correct erroneous bits. The placement and calculation of the parity bits follow a binary positional system (1, 2, 4, 8, 16, ...), which forms the core of the error detection and correction mechanism. For example, for an 8-bit word it needs four redundancy bits while for a

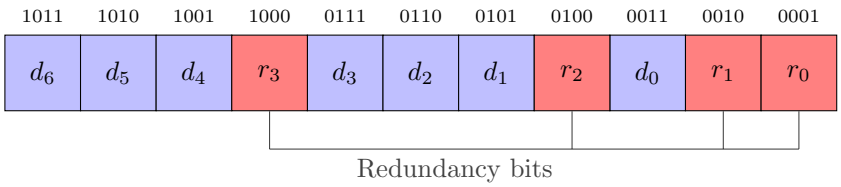


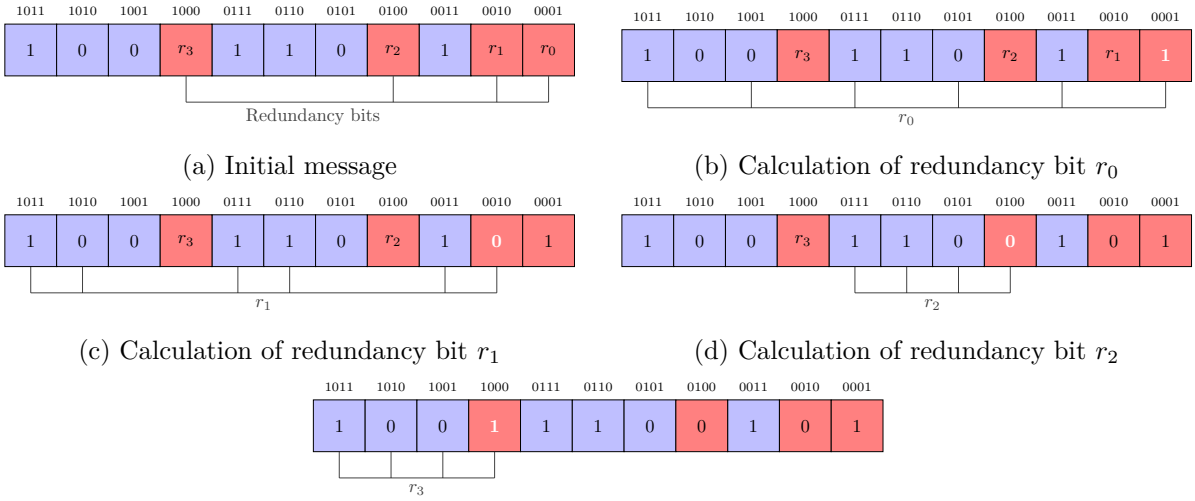
Figure 5.4: Hamming code (7,4) – functioning

32-bit word, it needs only 6 redundancy bits. By positioning the redundancy bits at the indexes of powers of two, it is then possible to correct an error if one is detected. Thus, for example, Hamming Code (11,7) owns seven bits data ($d_0 - d_6$) and four redundancy bits ($r_0 - r_3$). Data bits and redundancy bits are placed according to Figure 5.4. The most common Hamming Codes is the (7,4), which use four data bits and three redundancy bits. For this code, redundancy bits are computed according to Equation 5.2. This equation calculation is also represented in Figure 5.5. If the initial message to be sent is `0b1001101` in binary, the redundancy bit r_0 will be computed as $r_0 = d_0 \oplus d_1 \oplus d_3 \oplus d_4 \oplus d_6$. Thus, r_0 will be equals to `1` as depicted in Figure 5.5b. It is worth noting that this code is not fully used, because with four redundancy bits, Hamming code is able to protect up to eleven data bits to form Hamming Code (15,11).

$$\begin{aligned} r_0 &= d_0 \oplus d_1 \oplus d_3 \oplus d_4 \oplus d_6 \\ r_1 &= d_0 \oplus d_2 \oplus d_3 \oplus d_5 \oplus d_6 \\ r_2 &= d_1 \oplus d_2 \oplus d_3 \\ r_3 &= d_4 \oplus d_5 \oplus d_6 \end{aligned} \tag{5.2}$$

$$\begin{aligned} nr_0 &= r_0 \oplus d_0 \oplus d_1 \oplus d_3 \oplus d_4 \oplus d_6 = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 1 \\ nr_1 &= r_1 \oplus d_0 \oplus d_2 \oplus d_3 \oplus d_5 \oplus d_6 = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 1 \\ nr_2 &= r_2 \oplus d_1 \oplus d_2 \oplus d_3 = 0 \oplus 0 \oplus 1 \oplus 0 = 1 \\ nr_3 &= r_3 \oplus d_4 \oplus d_5 \oplus d_6 = 1 \oplus 0 \oplus 0 \oplus 1 = 0 \end{aligned} \tag{5.3}$$

Figure 5.6 presents an example of the detection and correction of an error. Figure 5.6a depicts the message sent `0b10011100101` (1253 in decimal). A fault occurs during the transmission in the bit d_3 (Figure 5.6b). The received message is `0b10010100101` (1189 in decimal). During the verification of the redundancy bits. The equation 5.3 shows how the new redundancy bits are calculated from the received redundancy and data bits. The association of these new redundancy bits ($nr_0 - nr_3$) is call the syndrome. This syndrome represents the position of the faulted bit and needs to be read backward from nr_3 to nr_0. As shown in Equation 5.3, the syndrome



(a) Initial message

(b) Calculation of redundancy bit r_0

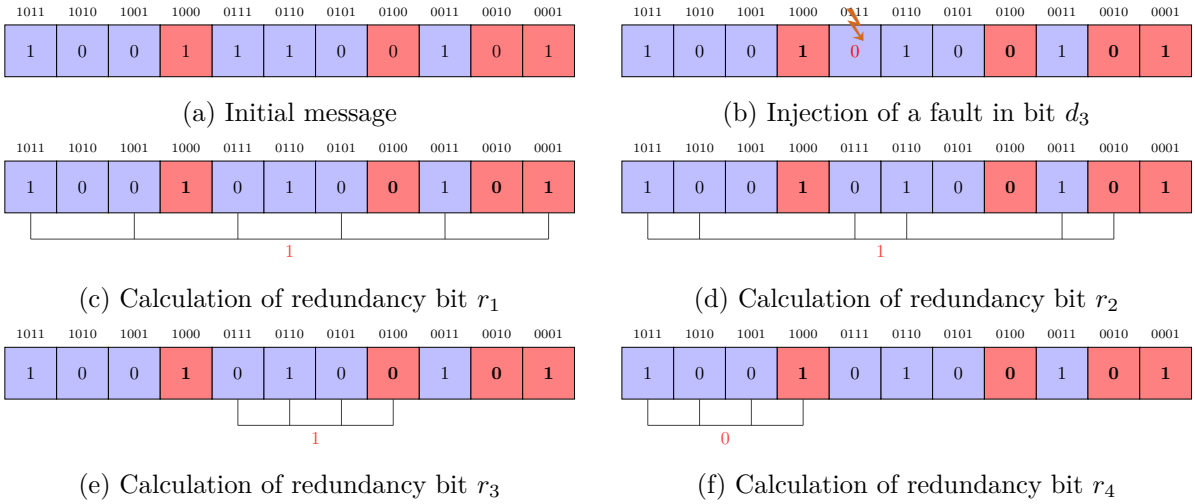
(c) Calculation of redundancy bit r_1

(d) Calculation of redundancy bit r_2

(e) Calculation of redundancy bit r_3

Figure 5.5: Hamming code (7,4) redundancy bits calculations

equals `0b0111`. This is the correct position of the fault that happened in Figure 5.6b. The same sequence is realised if a fault happen in a redundancy bit. This can be explained as each data bit is checked by at least two redundancy bits while a redundancy bit is checked only by itself during the decoding phase.



(a) Initial message

(b) Injection of a fault in bit d_3

(c) Calculation of redundancy bit r_1

(d) Calculation of redundancy bit r_2

(e) Calculation of redundancy bit r_3

(f) Calculation of redundancy bit r_4

Figure 5.6: Example of a faulted message with Hamming code (7,4)

Table 5.3: DIFT-related protected registers – Hamming code

	Protected register	Number of bits	Number of protected bits	Number of redundancy bits
Group 1	TCR	32	22	5
Group 2	TPR	32	22	5
Group 3	Register File (Tag)	32	32	6
Group 4	Tag destination address 16×1-bit registers	5	5	4
Group 5	3×2-bit registers 1×4-bit register	26	26	5
Total		127	107	25

5.4.2 Strategy: Minimisation of redundancy bits

In order to implement Hamming code, we used the same idea as the previous countermeasure: redundancy bits optimisation. We used the same five groups as depicted in Table 5.3. As we only protect 22 bits of the 32 bits from TCR and TPR registers, we only need 5 bits of redundancy, instead of 6 bits.

Figure 5.7 presents the proposed implementation for Hamming code. We do not integrate control signals for clarity. This implementation is straightforward. In order to protect a register or multiples independent registers, we choose to send the input(s) directly to both the protected register(s) (shown in blue) and an encoder. The encoder calculates the different redundancy bits using combinatorial logic, storing the resulting redundancy bits in a separate register, depicted in red in the figure. The redundancy bits are stored in this register at the same cycle as the input(s) value are stored in the protected register(s). Subsequently, the decoder computes the parity of the protected register and compares it with the redundancy bits stored in the redundancy bits register. If a difference is detected, it indicates the injection of a fault, which causes a signal to be sent to indicate the detection. But also, thanks to Hamming code, we are able to determine where the fault happened and so the decoder will correct the faulted value (dashed arrows). Then this corrected value will be sent to the pipeline, and at the same time, we correct the faulted register.

In order to protect the set of 32 1-bit registers from the Register File Tag, we rely on a slightly different approach. Figure 5.8 presents the second approach with six redundancy bits. We have developed a slightly different approach to minimise the impact on the original design of the D-RI5CY tag register file. Basically, we use the existing 2 input ports interfaces instead of adding a third input port dedicated to correction. We choose to send the input directly to both the protected register (shown in blue) and an encoder. As in the previous case, the decoder allows the detection of an error due to a bit-flip fault in one of the registers. With Hamming Code protection, the decoder produces corrected outputs (dashed arrows) which are propagated

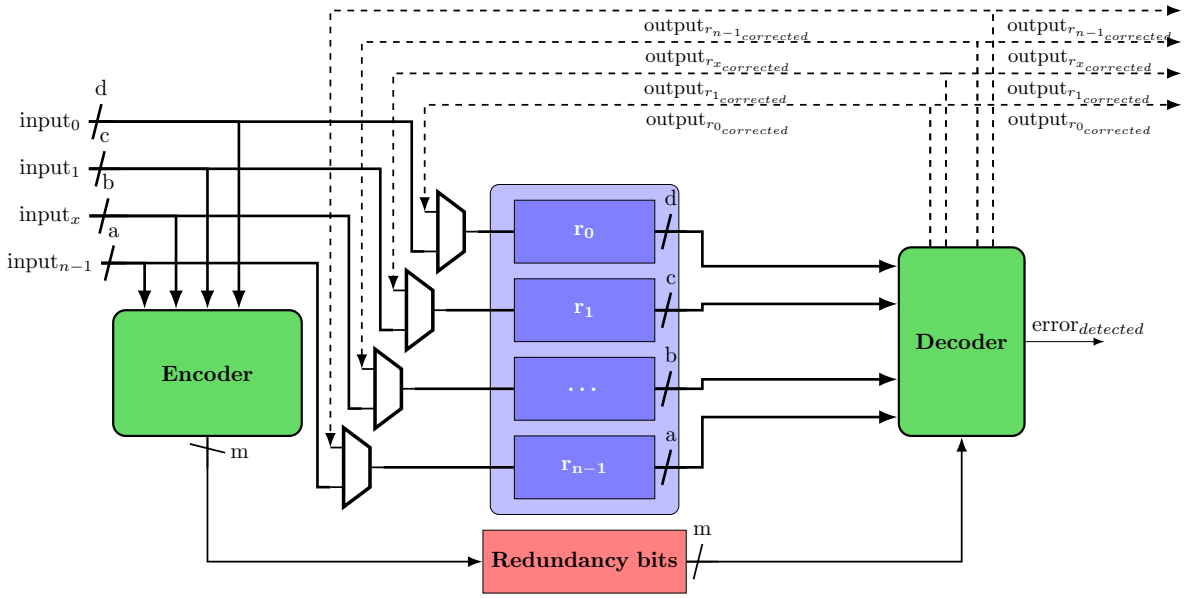


Figure 5.7: Implementation of Hamming Code

to the tag register outputs. If a fault is detected, the corrected output is forwarded to the tag register interface. As soon as one of the two input ports is available, this corrected value is stored in the faulty register to correct the detected fault. A fresh input value has priority on the corrected value to ensure the data flow correctness.

5.5 Hamming Codes - SECDED

William ▶*revoir entièrete du chapitre avec ajout SECDED*◀

5.5.1 Hamming Code - SECDED in a nutshell

5.5.2 Strategy: Minimisation of redundancy bits

5.6 Evaluation results

This section presents logical fault injection simulation results considering our two fault models: *single bit-flip in one register at a given clock cycle* and *single bit-flip in two registers at two clock cycles*. For protected implementations, faults are injected into both DIFT-related and protection-related registers.

Table 5.4 presents the results of the FPGA implementation using Vivado 2023.2, targeting the Xilinx Zynq-7000 of the Zedboard development board. It compares different protection mechanisms in terms of resource utilisation and maximum operating frequency. The table lists

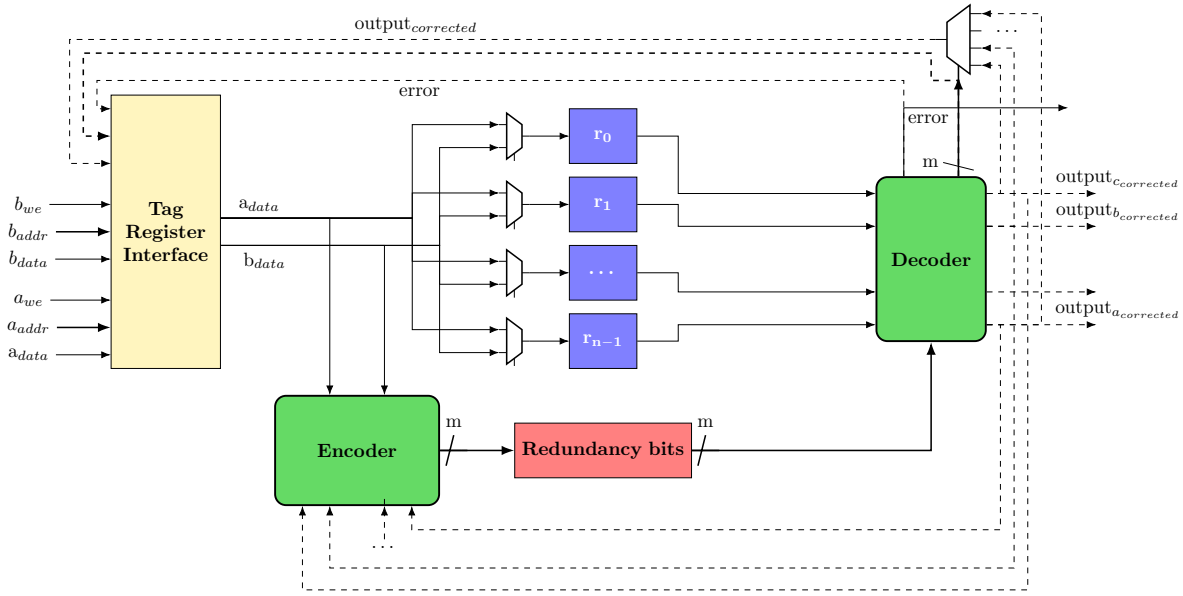


Figure 5.8: Implementation of Hamming Code – Register File Tag

Table 5.4: FPGA implementation results — Vivado 2023.2

Protection	Number of LUT	Number of FF	Maximum frequency
Baseline	6,597 (-4.54%)	2,211 (-5.31%)	49.10 MHz (3%)
D-RI5CY	6,911 (0%)	2,335 (0%)	47.60 MHz (0%)
Simple parity	7,011 (1.45%)	2,337 (0.09%)	47.60 MHz (0%)
Hamming Code SECDED	7,283 (5.38%)	2,361 (1.11%)	47.40 MHz (-0.36%)

the number of Look-Up Tables (LUT), the number of Flip-Flops (FFs), and the maximum achievable frequency for each protection scheme. The D-RI5CY mechanism serves as reference. The baseline version represents the processor without the DIFT protection, showing a reduction in both LUT and FF usage by 4.54% and 5.31%, respectively, while achieving a 3% improvement in maximum frequency compared to the D-RI5CY. Simple parity protection slightly increases LUT usage by 1.45%, with negligible impact on FFs and no change in the maximum frequency. The Hamming Code protection implementation introduces the most significant overhead, with a 5.38% increase in LUT and a 1.11% increase in FFs, alongside a minor reduction in maximum frequency by 0.36%. This comparison highlights the trade-offs between resource utilisation and performance across different protection mechanisms in FPGA implementations.

Now, we will compare these protections in terms of security. Regarding the "*single bit-flip in one register at a given clock cycle*" fault model, Table 5.5 shows the results obtained for the three considered use cases with and without protections. It is worth noting that we never get any crashes since we target the DIFT-related registers only. These registers do not impact

Table 5.5: Logical fault injection simulation campaigns results for single bit-flip in one register at a given clock cycle

		Crash	Silent	Delay	Detection	Detection & Correction	Success	Total	Execution time
Buffer overflow	No protection	0	738	12	—	—	12 (1.57%)	762	0:11
	Simple parity	0	0	0	792	—	0	792	0:08
	Hamming Code	0	0	0	—	912	0	912	0:12
Format String	No protection	0	946	41	—	—	29 (2.85%)	1,016	01:52
	Simple parity	0	0	0	1,056	—	0	1,056	01:30
	Hamming Code	0	0	0	—	1,216	0	1,216	01:50
Compare Compute	No protection	0	491	7	—	—	10 (1.97%)	508	0:02
	Simple parity	0	0	0	528	—	0	528	0:02
	Hamming Code	0	0	0	—	608	0	608	0:03
	SECDED	0			—				
Total							51	7,398	

Table 5.6: Logical fault injection simulation campaigns results for single bit-flip in two registers at two clock cycles

		Crash	Silent	Delay	Detection	Detection & Correction	Success	Total	Execution time
Buffer Overflow	No protection	0	238,633	1,143	—	—	2,159 (0.89%)	241,935	42:12
	Simple parity	0	0	0	261,360	—	0	261,360	64:24
	Hamming Code	0	0	0	—	346,560	0	346,560	66:48
Format String	No protection	0	429,260	12,192	—	—	10,160 (2.25%)	451,612	544:52
	Simple parity	0	0	0	487,872	—	0	487,872	389:20
	Hamming Code	0	0	0	—	646,912	0	646,912	1069:36
Compare Compute	No protection	0	90,432	2,795	—	—	3,547 (3.67%)	96,774	12:42
	Simple parity	0	0	0	104,544	—	0	104,544	13:36
	Hamming Code	0	0	0	—	138,624	0	138,624	20:32
Total							15,866	2,776,193	

the control or instruction flow of the processor. The results obtained without protection are from Chapter 3. We obtain 51 successes out of 2286 fault injection simulations. Conversely, when employing simple parity protection, none of the 2376 simulations result in success, as each single-fault in this fault model is detected, achieving a 100% detection rate. With simple parity, an error signal is generated, which can be intercepted by a software running in the system to handle the fault, potentially halting the application if necessary. In contrast, the Hamming code protection corrects the fault within the same cycle it occurs, without providing any direct indication to the attacker. The results from the Hamming code simulations also show 0 success, but this time 100% of the faults are corrected. This ensures the application continues running as if no fault occurred. From the attacker's perspective, the fault does not affect the system's behaviour in any way.

Table 5.6 presents the results obtained considering the "*single bit-flip in two registers at two*

clock cycles" fault model. We conducted 2,776,193 simulations to present the results of this new fault model. For each simulation, we choose two bits in the same register or two registers, and we choose two different cycles, then, we flip one bit at a first cycle and flip the other one at the other cycle. It shows that without any protection, 15,866 fault injections among 790,321 simulations (2.01%) lead to a successful attack in the three use case, while no successes are reported from Simple Parity or Hamming Code. Each fault is corrected thanks to Hamming Code.

5.7 Summary

In this chapter, we presented three countermeasures in order to protect the DIFT mechanism against fault injection attacks. For that, we considered two fault models: *single bit-flip in one register at a given clock cycle* and *single bit-flip in two registers at two clock cycles*. These fault models are used in real world fault injection attacks. The first countermeasure is based on parity code: simple parity and can be used to detect any errors. Thanks to this protection, we achieve a 100% fault detection in our considered fault model, but with the downside of giving an indication to the attacker as we emit a signal which can be caught by a running software to halt the application. In the other hand, we implemented a code-based protection: Hamming code. This protection is limited to only detection and correction of an error in our case. We propose two implementations. The first implementation is used to protect a set of registers together. The second implementation targets the protection of the *Register File Tag* with constraints such as the number of write ports available. Thanks to these implementations, we are able to handle 100% of the injected fault and correct them without any direct indication to the attacker. The third countermeasure is an Hamming Code with an additional parity bit, it is called SECDED. **William** ►*finir*◄ These three countermeasures give effective results against the fault models we have considered, while on the other hand, they have a limited impact on system performance and surface area.

In the next chapter, we will evaluate these protections against more complex fault models such as multi bit-flip fault and explore different implementation strategies in order to have a more robust protection against a wider range of attacks and fault models.

Chapter 6

Evaluation of groups composition and results

Contents

6.1 Introduction

The previous chapter presented two countermeasures against fault injection attacks and taking into account simple fault models, such as *single-bit flip inside one register at a given clock cycle.*

These countermeasures have been implemented by grouping the different DIFT-related registers in order to minimise the number of parity and redundancy bits. However, nowadays, studies [91, 174] have shown that is it possible to fault multiple bits precisely.

In this chapter, we present four different implementation's strategies of countermeasures to better protect the D-RI5CY mechanism against more complex fault models. Each implementation will be presented in their respective section. Additionnally, we implement another version of Hamming code to detect double-bit errors and correct single-bit errors, this method is called SECDED for *Single Error Correction, Double Error Detection*. We do not present simple parity in this chapter because, from the results of the previous chapter, it is better to use Hamming code as it can correct the fault instead of just emitting a fault detection signal according the difference in terms of area and performance overheads.

Section 6.2 introduces the different fault models considered. Then, section 6.4 explains the four different strategies for Hamming code and gives the results associated to the fault models. Section 6.5 introduces the SECDED countermeasure and gives the results associated to the five implementations. Finally in section 6.6, we compare the results obtained from these two countermeasures and evaluate them in terms of efficiency, performance, and area overhead.

6.2 Fault models considered in this chapter

In Chapter 5, we presented the results of fault injection campaigns targeting *a single bit-flip in one register at a given clock cycle*, and *a single bit-flip in two registers at two distinct clock cycles*. We demonstrated that lightweight countermeasures, such as simple parity or Hamming code, are effective in protecting our DIFT mechanism against single bit-flips occurring in one register at one clock cycle or in two registers at two clock cycles.

In this chapter, we extend our analysis to consider an attacker capable of injecting faults into DIFT-related registers, leading to *a single bit-flip in two registers at a given clock cycle*. Furthermore, we account for an attacker able to induce *multi-bit faults in one target at a given clock cycle* as well as *multi-bit faults in two targets at a given clock cycle*. These fault models, introduced in Chapter 4, are exhaustively tested across registers ranging from 1-bit to 6-bit in size. Registers larger than 10 bits, such as the configuration registers TPR and TCR, are excluded due to their size. For instance, simulating an exhaustive attack on a single 32-bit register for one cycle would require 2^{32} simulations (i.e: 4,294,967,296 simulations), and for two 32-bit registers, the number of simulations would reach $2^{32} \times 2^{32}$.

The two fault models are exhaustively simulated across all possible values of these registers. To meet this objective, any DIFT-related register that maintains a 1-bit tag value, drives tag propagation or tag update processes, or holds security policy configurations, can be targeted. Additionally, registers storing redundancy bits for protection mechanisms are also considered.

6.3 Simple Parity

6.3.1 Strategy 1: Minimisation of redundancy bits

6.4 Hamming Code

6.4.1 Strategy 1: Minimisation of redundancy bits

6.4.2 Strategy 2: Protection by pipeline stage

6.4.3 Strategy 3: Protection of all registers individually

6.4.4 Strategy 4: Protection of all registers individually with CSRs slicing

6.4.5 Strategy 5: Cooking spaghetti is not forbidden

6.4.6 Results

6.5 Hamming Code - SECDED

6.5.1 Strategy 1: Optimisation of redundancy bits

6.5.2 Strategy 2: Protection by pipeline stage

6.5.3 Strategy 3: Protection of all registers individually

6.5.4 Strategy 4: Protection of all registers individually with CSRs slicing

6.5.5 Strategy 5: Smart protection by pipeline stage

6.5.6 Results

6.6 Evaluation and discussion

6.7 Summary

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] Transforma Insights; Exploding Topics, *Number of Internet of Things (IoT) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033*, Online. Accessed 13 August 2024, 2024, URL: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.

[2] Transforma Insights, *Internet of Things (IoT) total annual revenue worldwide from 2020 to 2030*, Online. Accessed 13 August 2024, 2023, URL: https://www.statista.com/statistics/1194709/iot-revenue-worldwide/.

[3] Mardiana binti Mohamad Noor and Wan Haslina Hassan, "Current research on Internet of Things (IoT) security: A survey", *in*: *Computer Networks* 148 (2019), pp. 283–294, ISSN: 1389-1286, DOI: https://doi.org/10.1016/j.comnet.2018.11.025.

[4] Eryk Schiller et al., "Landscape of IoT security", *in*: *Computer Science Review* 44 (2022), p. 100467, ISSN: 1574-0137, DOI: https://doi.org/10.1016/j.cosrev.2022.100467.

[5] Jannatul Ferdous et al., "A Review of State-of-the-Art Malware Attack Trends and Defense Mechanisms", *in*: *IEEE Access* 11 (2023), pp. 121118–121141, DOI: 10.1109/ACCESS.2023.3328351.

[6] C. Cowan et al., "Buffer overflows: attacks and defenses for the vulnerability of the decade", *in*: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2, 2000, 119–129 vol.2, DOI: 10.1109/DISCEX.2000.821514.

[7] Bruno Dorsemaine et al., "A new approach to investigate IoT threats based on a four layer model", *in*: *2016 13th International Conference on New Technologies for Distributed Systems (NOTERE)*, 2016, pp. 1–6, DOI: 10.1109/NOTERE.2016.7745830.

[8] Lwin Khin Shar and Hee Beng Kuan Tan, "Defending against Cross-Site Scripting Attacks", *in*: *Computer* 45.*3* (2012), pp. 55–62, DOI: 10.1109/MC.2011.261.

[9] Mauro Conti, Nicola Dragoni, and Viktor Lesyk, "A Survey of Man In The Middle Attacks", *in*: *IEEE Communications Surveys & Tutorials* 18.*3* (2016), pp. 2027–2051, DOI: 10.1109/COMST.2016.2548426.

[10] Aikaterini Mitrokotsa, Melanie R. Rieback, and Andrew S. Tanenbaum, "Classifying RFID attacks and defenses", *in*: *Information Systems Frontiers* 12.*5* (2010), pp. 491–505, DOI: 10.1007/s10796-009-9210-z.

[11] Hossein Pirayesh and Huacheng Zeng, "Jamming Attacks and Anti-Jamming Strategies in Wireless Networks: A Comprehensive Survey", *in*: *IEEE Communications Surveys & Tutorials* 24.*2* (2022), pp. 767–809, DOI: 10.1109/COMST.2022.3159185.

[12] Mampi Devi and Abhishek Majumder, "Side-Channel Attack in Internet of Things: A Survey", *in*: *Applications of Internet of Things*, Singapore: Springer Singapore, 2021, pp. 213–222, ISBN: 978-981-15-6198-6, DOI: 10.1007/978-981-15-6198-6_20.

[13] Paul C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", *in*: *Advances in Cryptology — CRYPTO '96*, Springer Berlin Heidelberg, 1996, pp. 104–113, ISBN: 978-3-540-68697-2, DOI: 10.1007/3-540-68697-5_9.

[14] H. Bar-El et al., "The Sorcerer's Apprentice Guide to Fault Attacks", *in*: *Proceedings of the IEEE* 94.*2* (2006), pp. 370–382, DOI: 10.1109/JPROC.2005.862424.

[15] Alessandro Barenghi et al., "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures", *in*: *Proceedings of the IEEE* 100.*11* (2012), pp. 3056–3076, DOI: 10.1109/JPROC.2012.2188769.

[16] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman, "Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation", *in*: *Journal of Hardware and Systems Security* 2 (2018), pp. 111–130, DOI: 10.1007/s41635-018-0038-1.

[17] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults", *in*: *Advances in Cryptology — EUROCRYPT '97*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 37–51, ISBN: 978-3-540-69053-5, DOI: 10.1007/3-540-69053-0_4.

[18] 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/.

[19] Riscure, *Laser Station 2*, URL: https://www.riscure.com/products/laser-station-2/.

[20] NewAE, *ChipWhisperer*, URL: https://www.newae.com/chipwhisperer.

[21] NewAE, *ChipSHOUTER*, URL: https://www.newae.com/chipshouter.

[22] Martin S. Kelly and Keith Mayes, "High Precision Laser Fault Injection using Low-cost Components", *in*: *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 219–228, DOI: 10.1109/HOST45689.2020.9300265.

[23] Johan Laurent et al., "Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures", *in*: *Design, Automation & Test in Europe Conference (DATE)*, 2019, DOI: 10.23919/DATE.2019.8715158.

[24] Thomas Trouchkine et al., “Electromagnetic Fault Injection Against a Complex CPU, toward new Micro-architectural Fault Models”, *in*: *Journal of Cryptographic Engineering* (2021), DOI: 10.1007/s13389-021-00259-6.

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

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

[27] Xhani Marvin Saß, Richard Mitev, and Ahmad-Reza Sadeghi, “Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M”, *in*: *32nd USENIX Security Symposium (USENIX Security 23)*, USENIX Association, Aug. 2023, pp. 6239–6256, ISBN: 978-1-939133-37-3, DOI: 10.48550/arXiv.2302.06932.

[28] Shoei Nashimoto et al., “Bypassing Isolated Execution on RISC-V using Side-Channel-Assisted Fault-Injection and Its Countermeasure”, *in*: *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)* (2021), DOI: 10.46586/tches.v2022.i1.28-68.

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

[30] 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.

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

[32] Monica S. Lam et al., “Securing web applications with static and dynamic information flow tracking”, *in*: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Association for Computing Machinery, 2008, pp. 3–12, ISBN: 9781595939777, DOI: 10.1145/1328408.1328410.

[33] Andrew Ferraiuolo et al., “Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis”, *in*: *SIGARCH Comput. Archit. News* 45.*1* (Apr. 2017), pp. 555–568, ISSN: 0163-5964, DOI: 10.1145/3093337.3037739.

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

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] Carlton Shepherd et al., “Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis”, *in*: *Computers & Security* 111 (2021), ISSN: 0167-4048, DOI: 10.1016/j.cose.2021.102471.

[58] Shahed E. Quadir et al., “A Survey on Chip to System Reverse Engineering”, *in*: *J. Emerg. Technol. Comput. Syst.* 13.*1* (Apr. 2016), ISSN: 1550-4832, DOI: 10.1145/2755563.

[59] Marc Fyrbiak et al., “Hardware reverse engineering: Overview and open challenges”, *in*: *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, 2017, pp. 88–94, DOI: 10.1109/IVSW.2017.8031550.

[60] Jeffrey Friedman, “TEMPEST: A Signal Problem”, *in*: 2 (1972), URL: https://www.nsa.gov/portals/75/documents/news-features/declassified-documents/cryptologic-spectrum/tempest.pdf.

[61] Paul Kocher, Joshua Jaffe, Benjamin Jun, et al., “Introduction to differential power analysis and related attacks”, *in*: (1998).

[62] Paul Kocher et al., “Introduction to differential power analysis”, *in*: *Journal of Cryptographic Engineering* 1 (2011), pp. 5–27, DOI: 10.1007/s13389-011-0006-y.

[63] Louis Goubin and Jacques Patarin, “DES and Differential Power Analysis The "Duplication" Method”, *in*: *Cryptographic Hardware and Embedded Systems*, Springer Berlin Heidelberg, 1999, pp. 158–172, ISBN: 978-3-540-48059-4, DOI: 10.1007/3-540-48059-5_15.

[64] Moritz Lipp et al., “PLATYPUS: Software-based Power Side-Channel Attacks on x86”, *in*: *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 355–371, DOI: 10.1109/SP40001.2021.00063.

[65] David Brumley and Dan Boneh, “Remote timing attacks are practical”, *in*: *Computer Networks* 48.*5* (2005), Web Security, pp. 701–716, ISSN: 1389-1286, DOI: https://doi.org/10.1016/j.comnet.2005.01.010.

[66] Asanka Sayakkara, Nhien-An Le-Khac, and Mark Scanlon, “A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics”, *in*: *Digital Investigation* 29 (2019), pp. 43–54, ISSN: 1742-2876, DOI: 10.1016/j.diin.2019.03.002.

[67] Johann Heyszl et al., "Localized Electromagnetic Analysis of Cryptographic Implementations", *in*: *Topics in Cryptology – CT-RSA 2012*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 231–244, ISBN: 978-3-642-27954-6, DOI: 10.1007/978-3-642-27954-6_15.

[68] Amit Kumar et al., "Efficient simulation of EM side-channel attack resilience", *in*: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 123–130, DOI: 10.1109/ICCAD.2017.8203769.

[69] Christian Wittke, Zoya Dyka, and Peter Langendoerfer, "Comparison of EM Probes Using SEMA of an ECC Design", *in*: *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2016, pp. 1–5, DOI: 10.1109/NTMS.2016.7792439.

[70] Jiaji He et al., "EM Side Channels in Hardware Security: Attacks and Defenses", *in*: *IEEE Design & Test* 39.*2* (2022), pp. 100–111, DOI: 10.1109/MDAT.2021.3135324.

[71] Jean-Jacques Quisquater and David Samyde, "ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards", *in*: *Smart Card Programming and Security*, Springer Berlin Heidelberg, 2001, pp. 200–210, ISBN: 978-3-540-45418-2, DOI: 10.1007/3-540-45418-7_17.

[72] Michael Hutter and Jörn-Marc Schmidt, "The Temperature Side Channel and Heating Fault Attacks", *in*: *Smart Card Research and Advanced Applications*, Cham: Springer International Publishing, 2014, pp. 219–235, ISBN: 978-3-319-08302-5, DOI: 10.1007/978-3-319-08302-5_15.

[73] Abdullah Aljuffri et al., "Applying Thermal Side-Channel Attacks on Asymmetric Cryptography", *in*: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29.*11* (2021), pp. 1930–1942, DOI: 10.1109/TVLSI.2021.3111407.

[74] Michael Backes et al., "Acoustic side-channel attacks on printers", *in*: *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, Washington, DC: USENIX Association, 2010, p. 20, DOI: 10.5555/1929820.1929847.

[75] Daniel Genkin, Adi Shamir, and Eran Tromer, "Acoustic Cryptanalysis", *in*: *Journal of Cryptology* 30 (2017), pp. 392–443, DOI: 10.1007/s00145-015-9224-2.

[76] Joshua Harrison, Ehsan Toreini, and Maryam Mehrnezhad, "A Practical Deep Learning-Based Acoustic Side Channel Attack on Keyboards", *in*: *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2023, pp. 270–280, DOI: 10.1109/EuroSPW59978.2023.00034.

[77] D. Binder, E. C. Smith, and A. B. Holman, "Satellite Anomalies from Galactic Cosmic Rays", *in*: *IEEE Transactions on Nuclear Science* 22.*6* (1975), pp. 2675–2680, DOI: 10.1109/TNS.1975.4328188.

[78] J. F. Ziegler, "Terrestrial cosmic rays", *in*: *IBM Journal of Research and Development* 40.*1* (1996), pp. 19–39, DOI: 10.1147/rd.401.0019.

[79] J. F. Ziegler and W. A. Lanford, "Effect of Cosmic Rays on Computer Memories", *in*: *Science* 206.*4420* (1979), pp. 776–788, DOI: 10.1126/science.206.4420.776.

[80] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations", *in*: *Journal of Cryptology* 14 (2001), pp. 101–119, DOI: 10.1007/s001450010016.

[81] Haissam Ziade, Rafic Ayoubi, and Raoul Velazco, "A survey on Fault Injection Techniques", *in*: *The international Arab journal of information technology* 1.*2* (Jan. 2004), pp. 171–186, URL: https://hal.science/hal-00105562.

[82] Roberta Piscitelli, Shivam Bhasin, and Francesco Regazzoni, "Fault attacks, injection techniques and tools for simulation", *in*: *2015 10th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2015, pp. 1–6, DOI: 10.1109/DTIS.2015.7127352.

[83] Jakub Breier and Xiaolu Hou, "How Practical Are Fault Injection Attacks, Really?", *in*: *IEEE Access* 10 (2022), pp. 113122–113130, DOI: 10.1109/ACCESS.2022.3217212.

[84] Wikipedia contributors, *Decapping— Wikipedia*, [Online; accessed 26-August-2024], 2019, URL: https://en.wikipedia.org/wiki/Decapping.

[85] Sergei P. Skorobogatov and Ross J. Anderson, "Optical Fault Induction Attacks", *in*: *Cryptographic Hardware and Embedded Systems*, Springer Berlin Heidelberg, 2002, pp. 2–12, ISBN: 978-3-540-36400-9, DOI: 10.1007/3-540-36400-5_2.

[86] Jörn-Marc Schmidt and Michael Hutter, "Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results", *in*: *Austrochip 2007 : 15th Austrian Workhop on Microelectronics*, Verlag der Technischen Universität Graz, 2007, pp. 61–67, ISBN: 978-3-902465-87-0, URL: https://graz.elsevierpure.com/en/publications/optical-and-em-fault-attacks-on-crt-based-rsa-concrete-results.

[87] Oscar M. Guillen, Michael Gruber, and Fabrizio De Santis, "Low-Cost Setup for Localized Semi-invasive Optical Fault Injection Attacks", *in*: *Constructive Side-Channel Analysis and Secure Design*, Springer International Publishing, 2017, pp. 207–222, ISBN: 978-3-319-64647-3, DOI: 10.1007/978-3-319-64647-3_13.

[88] Ray Beaulieu et al., "The SIMON and SPECK Families of Lightweight Block Ciphers", *in*: (2013), URL: https://eprint.iacr.org/2013/404.

[89] Jean-Max Dutertre et al., "Laser Fault Injection at the CMOS 28 nm Technology Node: an Analysis of the Fault Model", *in*: *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2018, pp. 1–6, DOI: 10.1109/FDTC.2018.00009.

[90] Brice Colombier et al., "Laser-induced Single-bit Faults in Flash Memory: Instructions Corruption on a 32-bit Microcontroller", *in*: *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 1–10, DOI: 10.1109/HST.2019.8741030.

[91] 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.

[92] Breier Jakub et al., "Attacks in Reality: the Limits of Concurrent Error Detection Codes Against Laser Fault Injection", *in*: *Journal of Hardware and Systems Security* 1 (Dec. 2017), DOI: 10.1007/s41635-017-0020-3.

[93] Sara Faour et al., "Implications of Physical Fault Injections on Single Chip Motes", *in*: *2023 IEEE 9th World Forum on Internet of Things (WF-IoT)*, 2023, pp. 1–6, DOI: 10.1109/WF-IoT58464.2023.10539380.

[94] Randy Torrance and Dick James, "The State-of-the-Art in IC Reverse Engineering", *in*: *Cryptographic Hardware and Embedded Systems - CHES 2009*, Springer Berlin Heidelberg, 2009, pp. 363–381, ISBN: 978-3-642-04138-9, DOI: 10.1007/978-3-642-04138-9_26.

[95] Wikipedia contributors, *Focused Ion Beam — Wikipedia*, [Online; accessed 01-September-2024], 2024, URL: https://en.wikipedia.org/wiki/Focused_ion_beam.

[96] Stéphanie Anceau et al., "Nanofocused X-Ray Beam to Reprogram Secure Circuits", *in*: *Cryptographic Hardware and Embedded Systems – CHES 2017*, Springer International Publishing, 2017, pp. 175–188, ISBN: 978-3-319-66787-4, DOI: 10.1007/978-3-319-66787-4_9.

[97] S. Bouat et al., "X ray nanoprobe for fault attacks and circuit edits on 28-nm integrated circuits", *in*: *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2023, pp. 1–6, DOI: 10.1109/DFT59622.2023.10313553.

[98] 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.

[99] NewAE, *ChipWhisperer*, Online. Accessed 11 September 2024, URL: http://wiki.newae.com/V4:Tutorial_A2_Introduction_to_Glitch_Attacks_(including_Glitch_Explorer).

[100] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede, "An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs", *in*: *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2011, pp. 105–114, DOI: 10.1109/FDTC.2011.9.

[101] Alessandro Barenghi et al., "Low Voltage Fault Attacks on the RSA Cryptosystem", *in*: *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2009, pp. 23–31, DOI: 10.1109/FDTC.2009.30.

[102] Niek Timmers and Cristofaro Mune, "Escalating Privileges in Linux Using Voltage Fault Injection", *in*: *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2017, pp. 1–8, DOI: 10.1109/FDTC.2017.16.

[103] Nikolaos Athanasios Anagnostopoulos et al., "Low-Temperature Data Remanence Attacks Against Intrinsic SRAM PUFs", *in*: *2018 21st Euromicro Conference on Digital System Design (DSD)*, 2018, pp. 581–585, DOI: 10.1109/DSD.2018.00102.

[104] Riscure, *EM-FI Transient Probe with Adjustable Pulse Width*, URL: https://www.riscure.com/em-fi-transient-probe-apw/.

[105] Amine Dehbaoui et al., "Electromagnetic Glitch on the AES Round Counter", *in*: *Constructive Side-Channel Analysis and Secure Design*, Springer Berlin Heidelberg, 2013, pp. 17–31, ISBN: 978-3-642-40026-1, DOI: 10.1007/978-3-642-40026-1_2.

[106] Aakash Gangolli, Qusay H. Mahmoud, and Akramul Azim, "A Systematic Review of Fault Injection Attacks on IoT Systems", *in*: *Electronics* 11.*13* (2022), ISSN: 2079-9292, DOI: 10.3390/electronics11132023.

[107] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede, "Hardware Designer's Guide to Fault Attacks", *in*: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.*12* (2013), pp. 2295–2306, DOI: 10.1109/TVLSI.2012.2231707.

[108] Martin Otto, "Fault Attacks And Countermeasures", PhD thesis, University of Paderborn, 2005.

[109] Johannes Blömer and Jean-Pierre Seifert, "Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)", *in*: *Financial Cryptography*, Springer Berlin Heidelberg, 2003, pp. 162–181, ISBN: 978-3-540-45126-6, DOI: 10.1007/978-3-540-45126-6_12.

[110] Pei Luo et al., "Differential Fault Analysis of SHA3-224 and SHA3-256", *in*: *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016, pp. 4–15, DOI: 10.1109/FDTC.2016.17.

[111] Alexandre Menu et al., "Experimental Analysis of the Electromagnetic Instruction Skip Fault Model", *in*: *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2020, pp. 1–7, DOI: 10.1109/DTIS48698.2020.9081261.

[112] Maxime Madau et al., "The Impact of Pulsed Electromagnetic Fault Injection on True Random Number Generators", *in*: *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2018, pp. 43–48, DOI: 10.1109/FDTC.2018.00015.

[113] Wei He, Jakub Breier, and Shivam Bhasin, "Cheap and Cheerful: A Low-Cost Digital Sensor for Detecting Laser Fault Injection Attacks", *in*: *Security, Privacy, and Applied Cryptography Engineering*, Springer International Publishing, 2016, pp. 27–46, ISBN: 978-3-319-49445-6, DOI: 10.1007/978-3-319-49445-6_2.

[114] David El-Baze, Jean-Baptiste Rigaud, and Philippe Maurine, "A fully-digital EM pulse detector", *in*: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 439–444, URL: https://ieeexplore.ieee.org/abstract/document/7459351.

[115] Md Rafid Muttaki et al., "FTC: A Universal Sensor for Fault Injection Attack Detection", *in*: *2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2022, pp. 117–120, DOI: 10.1109/HOST54066.2022.9840177.

[116] Alessandro Barenghi et al., "Countermeasures against fault attacks on software implemented AES: effectiveness and cost", *in*: *Proceedings of the 5th Workshop on Embedded Systems Security*, WESS '10, Scottsdale, Arizona: Association for Computing Machinery, 2010, ISBN: 9781450300780, DOI: 10.1145/1873548.1873555.

[117] Nikolaus Theißing et al., "Comprehensive analysis of software countermeasures against fault attacks", *in*: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013, pp. 404–409, DOI: 10.7873/DATE.2013.092.

[118] Thomas Chamelot, Damien Couroussé, and Karine Heydemann, "SCI-FI: Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks", *in*: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 556–559, DOI: 10.23919/DATE54114.2022.9774685.

[119] Johan Laurent et al., "Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor", *in*: *Microprocessors and Microsystems* 71 (2019), p. 102862, ISSN: 0141-9331, DOI: 10.1016/j.micpro.2019.102862.

[120] Robert Schilling, Mario Werner, and Stefan Mangard, "Securing conditional branches in the presence of fault attacks", *in*: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 1586–1591, DOI: 10.23919/DATE.2018.8342268.

[121] Francisco Eugenio Potestad-Ordóñez et al., "Hardware Countermeasures Benchmarking against Fault Attacks", *in*: *Applied Sciences* 12.*5* (2022), ISSN: 2076-3417, DOI: 10.3390/app12052443.

[122] Marc Joye, Pascal Manet, and Jean-Baptiste Rigaud, "Strengthening hardware AES implementations against fault attacks.", *in*: *IET Inf. Secur.* 1.*3* (2007), pp. 106–110, DOI: 10.1049/iet-ifs_20060163.

[123] Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre, "On-Line Self-Test of AES Hardware Implementations", *in*: *DSN'07: Workshop on Dependable and Secure Nanocomputing*, June 2007, URL: https://hal-lirmm.ccsd.cnrs.fr/lirmm-00163405.

[124] G. Di Natale et al., "A Reliable Architecture for the Advanced Encryption Standard", *in*: *2008 13th European Test Symposium*, 2008, pp. 13–18, DOI: 10.1109/ETS.2008.26.

[125] Jeyavijayan Rajendran et al., "SLICED: Slide-based concurrent error detection technique for symmetric block ciphers", *in*: *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2010, pp. 70–75, DOI: 10.1109/HST.2010.5513109.

[126] P. Maistri, P. Vanhauwaert, and R. Leveugle, "A Novel Double-Data-Rate AES Architecture Resistant against Fault Injection", *in*: *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, 2007, pp. 54–61, DOI: 10.1109/FDTC.2007.8.

[127] Xiaofei Guo and Ramesh Karri, "Recomputing with Permuted Operands: A Concurrent Error Detection Approach", *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.*10* (2013), pp. 1595–1608, DOI: 10.1109/TCAD.2013.2263037.

[128] Noura Ait Manssour et al., "Processor Extensions for Hardware Instruction Replay against Fault Injection Attacks", *in*: *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2022, pp. 26–31, DOI: 10.1109/DDECS54261.2022.9770170.

[129] Charalampos Ananiadis et al., "On the development of a new countermeasure based on a laser attack RTL fault model", *in*: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 445–450, URL: https://ieeexplore.ieee.org/document/7459352.

[130] Hassen Mestiri et al., "A hardware FPGA implementation of fault attack countermeasure", *in*: *2014 15th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, 2014, pp. 178–183, DOI: 10.1109/STA.2014.7086674.

[131] G. Bertoni et al., "Error analysis and detection procedures for a hardware implementation of the advanced encryption standard", *in*: *IEEE Transactions on Computers* 52.*4* (2003), pp. 492–505, DOI: 10.1109/TC.2003.1190590.

[132] F. E. Potestad-Ordóñez et al., "Hamming-Code Based Fault Detection Design Methodology for Block Ciphers", *in*: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5, DOI: 10.1109/ISCAS45731.2020.9180451.

[133] Alexander Dörflinger et al., "ECC Memory for Fault Tolerant RISC-V Processors", *in*: *Architecture of Computing Systems – ARCS 2020*, Cham: Springer International Publishing, 2020, pp. 44–55, ISBN: 978-3-030-52794-5, DOI: 10.1007/978-3-030-52794-5_4.

[134] Chih-Hsu Yen and Bing-Fei Wu, "Simple error detection methods for hardware implementation of Advanced Encryption Standard", *in*: *IEEE Transactions on Computers* 55.*6* (2006), pp. 720–731, DOI: 10.1109/TC.2006.90.

[135] 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.

[136] ETH Zurich and Pulp Platform, *A Hardware Dynamic Information Flow Tracking Architecture for Low-level Security on a RISC-V Core*, 2016, URL: https://github.com/pulp-platform/pulpino.

[137] ETH Zurich and Pulp Platform, *PULP Platform - Open hardware, the way it should be!*, 2016, URL: https://pulp-platform.org/.

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

[139] 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.

[140] William Pensec, Vianney Lapôtre, and Guy Gogniat, "Scripting the Unpredictable: Automate Fault Injection in RTL Simulation for Vulnerability Assessment", *in*: *2024 27th Euromicro Conference on Digital System Design (DSD)*, 2024.

[141] Mirgita Frasheri et al., "Fault Injecting Co-simulations for Safety", *in*: *2021 5th International Conference on System Reliability and Safety (ICSRS)*, 2021, pp. 6–13, DOI: 10.1109/ICSRS53853.2021.9660728.

[142] 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.

[143] 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.

[144] 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.

[145] 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.

[146] 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.

[147] 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.

[148] 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.

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

[150] 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.

[151] Nasr-eddine Ouldei Tebina et al., "Ray-Spect: Local Parametric Degradation for Secure Designs: An application to X-Ray Fault Injection", *in*: *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2023, pp. 1–7, DOI: 10.1109/IOLTS59296.2023.10224894.

[152] Huanyu Wang et al., "SoFI: Security Property-Driven Vulnerability Assessments of ICs Against Fault-Injection Attacks", *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.*3* (2022), pp. 452–465, DOI: 10.1109/TCAD.2021.3063998.

[153] Jacob Grycel and Patrick Schaumont, "SimpliFI: Hardware Simulation of Embedded Software Fault Attacks", *in*: *Cryptography* 5.*2* (2021), ISSN: 2410-387X, DOI: 10.3390/cryptography5020015.

[154] Max Hoffmann, Falk Schellenberg, and Christof Paar, "ARMORY: Fully Automated and Exhaustive Fault Simulation on ARM-M Binaries", *in*: *IEEE Transactions on Information Forensics and Security* 16 (2021), pp. 1058–1073, DOI: 10.1109/TIFS.2020.3027143.

[155] Kit Murdock, Martin Thompson, and David Oswald, "FaultFinder: lightning-fast, multi-architectural fault injection simulation", *in*: *ASHES '24: Proceedings of the 2024 Workshop on Attacks and Solutions in Hardware Security*, Not yet published as of 09/09/2024; 8th Workshop on Attacks and Solutions in Hardware Security, ASHES 2024 ; Conference date: 18-10-2024 Through 18-10-2024, Association for Computing Machinery (ACM), Oct. 2024, DOI: 10.1145/3689939.3695788.

[156] 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.

[157] 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.

[158] 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.

[159] 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.

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

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

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

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

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

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

[166] Microsemi, *Modelsim reference manual 10.4c*, URL: https://www.microsemi.com/document-portal/doc_view/136364-modelsim-me-10-4c-command-reference-manual-for-libero-soc-v11-7.

[167] Xilinx, *Vivado reference manual 2023.2*, URL: https://docs.xilinx.com/r/en-US/ug835-vivado-tcl-commands/add_force.

[168] William Pensec et al., "Defending the Citadel: Fault Injection Attacks against Dynamic Information Flow Tracking and Related Countermeasures", *in*: *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Knoxville, United States, July 2024, URL: https://hal.science/hal-04620057.

[169] Loïc Zussa et al., "Investigation of timing constraints violation as a fault injection means", *in*: *27th Conference on Design of Circuits and Integrated Systems (DCIS)*, Avignon, France, Nov. 2012, URL: https://hal-emse.ccsd.cnrs.fr/emse-00742652.

[170] Franck Courbon et al., "Adjusting Laser Injections for Fully Controlled Faults", *in*: *Constructive Side-Channel Analysis and Secure Design*, Cham: Springer International Publishing, 2014, pp. 229–242, ISBN: 978-3-319-10175-0, DOI: 10.1007/978-3-319-10175-0_16.

[171] ALPhANOV, *Double Laser Fault Injection Microscope – D-LMS*, [Online; accessed 23-September-2024], URL: https://www.alphanov.com/en/products-services/double-laser-fault-injection.

[172] ALPhANOV, *ALPhANOV has designed a four-point laser rig for laser fault injections on integrated circuits.* [Online; accessed 23-September-2024], 2019, URL: https://www.alphanov.com/en/news/alphanov-has-designed-four-point-laser-rig-laser-fault-injections-integrated-circuits.

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

[174] Raphael Viera et al., "Tampering with the flash memory of microcontrollers: permanent fault injection via laser illumination during read operations", *in*: *Journal of Cryptographic Engineering* 14 (2024), pp. 207–221, DOI: 10.1007/s13389-023-00335-z.

COLLEGE MATHSTIC
DOCTORAL BRETAGNE
BRETAGNE OCEANE

Université Bretagne Sud

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.