Présentation de mes travaux de recherche – Sécurité Matérielle - IA - Contremesures

Présentation CNFM - Mastère SECNUM

William PENSEC

Maître de Conférences
LIRMM – Université de Montpellier
Montpellier, France

15 Décembre 2025

Objectifs de la séance - travaux de recherche

Présentation de mes travaux de recherche : Thèse de Doctorat

- Sécurité matérielle
- Attaques physiques : Injection de fautes
- Combinaison d'attaques logicielles et physiques en parallèle
- Étude de différentes countremesures

$\Rightarrow$ Démonstration à la fin de la séance par Ali Ait Hassou

Présentation de mes travaux de recherche : Sécurité des réseaux de neurones

- Sécurité matérielle
- Attaques physiques : Injection de fautes
- Combinaison d'attaques logicielles et physiques en parallèle
- Étude de différentes countremesures

Some examples of software attacks - example n°1

First software attack

- Created in 1988, worm distributed via internet
- Buffer overflow, using *finger* network service
- Exploit of weak passwords
- Automatic propagation
- Around 6,000 UNIX machines were infected.

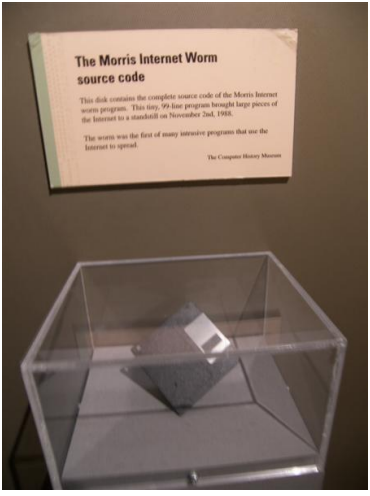


Figure 1: Floppy disk containing the source code for the Morris Worm, at the Computer History Museum

Some examples of software attacks - example n°2

Heartbleed - 2014

- Vulnerability in OpenSSL (Heartbeat extension)
- Out-of-bounds reading of server memory
- Leakage of private keys, passwords, sensitive data
- Remote attack, without authentication
- Affects a large part of the secure web (TLS) - $\approx$ 17% of secure server at that time (including: Wikipedia, Reddit, Pinterest, Amazon Web Services, GitHub, ...)

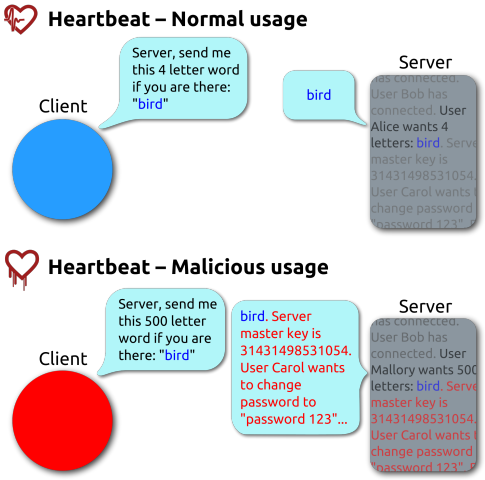


Figure 2: A depiction of Heartbleed.

WannaCry - 2017

- Self-propagating ransomware.
- Exploits a vulnerability (EternalBlue - exploit developed by the NSA) in Windows.
- Spreads without user interaction.
- Paralyses hospitals, businesses and government agencies.
- Over 300,000 machines infected in just a few days (including: FedEx, Renault, Vodafone, Telefónica).

Figure 3: Screenshot of the ransom note left on an infected system

From software to hardware

- Software attacks exploit code errors.
- But software assumes reliable, silent hardware.
- In practice, hardware:
  - leaks information
  - makes mistakes
- These phenomena are physically observable and exploitable.

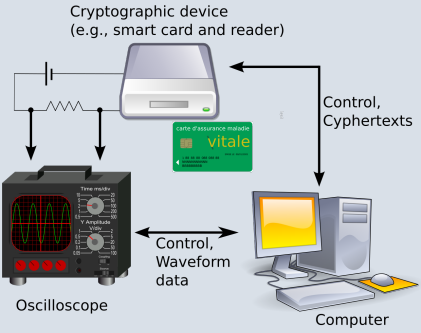
Some examples of hardware attacks - example n°1

Meltdown

- 2018 - Google and Researchers in TU Graz.
- A non-privileged programme runs on the processor.
- The processor performs unauthorised speculative memory accesses.
- The data is then observed via a timing leak channel (cache).
- Consequences:
  - Reading kernel memory from user space
  - Leakage of sensitive data (keys, passwords, system information)
  - Breach of privilege isolation
  - Impact on shared systems (servers, cloud)

Side-channel — Differential Power Analysis (DPA)

- Analysis of power consumption.
- Correlation with internal operations.
- Extraction of cryptographic keys (DES, AES).
- Demonstrated on smart cards and embedded systems.
- Consequences:
  - Extraction of the secret key,
  - Possibility of cloning the card,
  - Identity theft or financial fraud.



Some examples of hardware attacks - example n°3

Fault Injection

- Bypassing checks, extracting secrets.
- Clock or voltage glitching, laser, EMFI.
- A hardware disruption causes a calculation error.
- Consequences:
  - Bypassing security mechanisms,
  - Execution of unauthorised code,
  - Extraction of cryptographic keys,
  - Total compromise of the system.

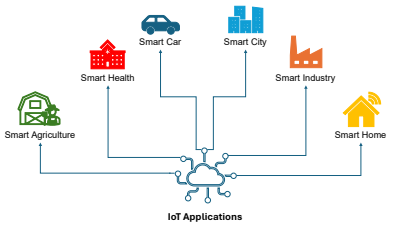
Outline

Part I

I. Introduction

Internet of Things (IoT)

- Wide range of application
- Fast growing market
- Rely on sensors, depending on their applications
- Collect and share data
- Manipulation of sensitive data
- Increasingly vulnerable to multiple threats



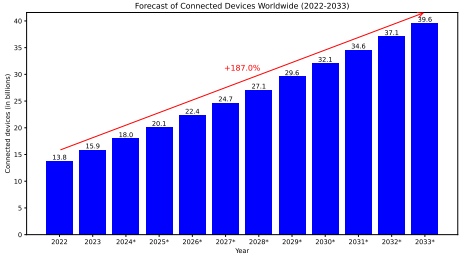


Figure 4: From [1]

Threats

- Network threats: Man-In-The-Middle [2], jamming [3], DoS, etc
- Software threats: memory overflow attacks [4], code execution, SQL injection, etc
- Hardware threats: Reverse Engineering, Side-Channel Attacks [5], Fault Injection Attacks [6]

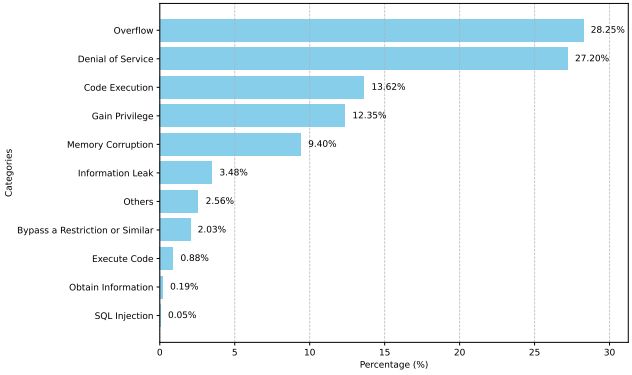


Figure 5: Data from BitDefender [7]

Threats

- Network threats: Man-In-The-Middle [2], jamming [3], DoS, etc
- **Software threats**: memory overflow attacks [4], code execution, SQL injection, etc
- **Hardware threats**: Reverse Engineering, Side-Channel Attacks [5], Fault Injection Attacks [6]

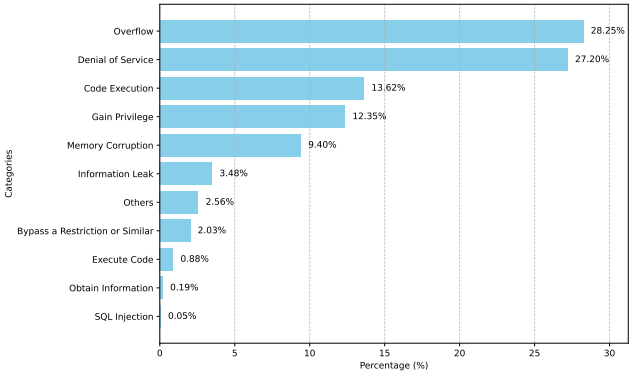


Figure 5: Data from BitDefender [7]

Software protection mechanisms

- Generic mechanisms to strengthen programme delivery
- Affect the control flow and/or the data flow

Mechanism	**Main objective**
Control-Flow Integrity (CFI)	Preventing control flow diversions
Shadow Stack	Protecting return addresses
Software Fault Isolation (SFI)	Illegitimate memory accesses
Dynamic Information Flow Tracking (DIFT)	Monitor and control the use of sensitive data

Key idea

These mechanisms assume correct execution of the underlying hardware.

Software protection mechanisms

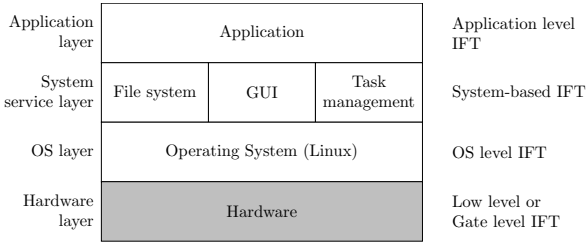
Mechanism	Control Flow	Data	Leaks Blocked	Type of Attack Blocked
CFI	✓	✗	✗	ROP, JOP, control-flow hijacking
Shadow Stack	✓(returns)	✗	✗	Stack corruption, fake returns
SFI	Memory access	✗	✗	Unauthorized memory access
DIFT	(indirect)	✓	✓	Data leaks, “data-only” attacks

Software threats: Dynamic Information Flow Tracking

- Security mechanism
- Protection against software attacks [8] (e.g.: *buffer overflow*, *format string*, *SQL injections*)
- Follow a security policy

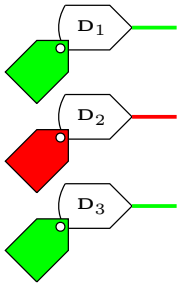
Different layers

- Application layer (software-based DIFT)
- System layer / OS layer (system-based DIFT)
- Hardware layer (hardware-based DIFT)
- Software Hardware co-design-based DIFT exist



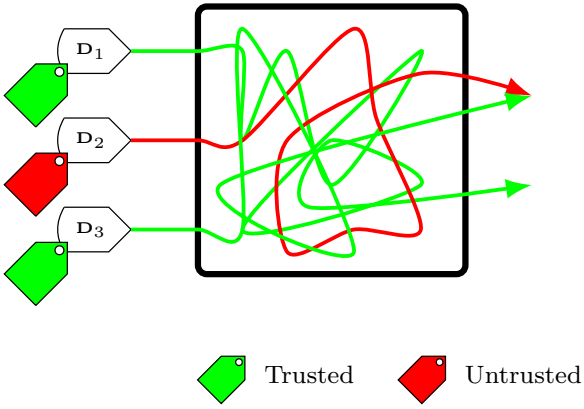
Three steps

- Tag initialisation



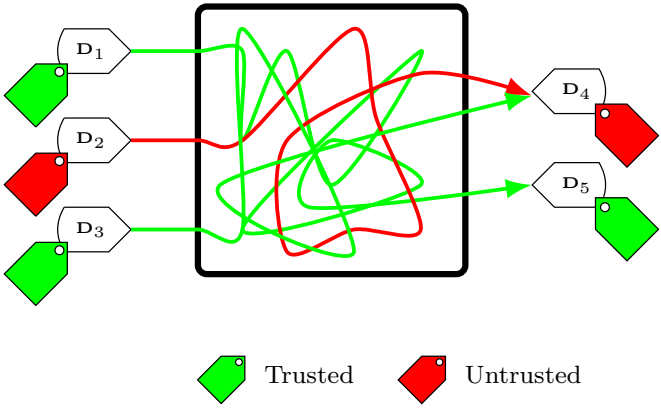
Three steps

- Tag initialisation
- Tag propagation



Three steps

- Tag initialisation
- Tag propagation
- Tag check



Software threats: Dynamic Information Flow Tracking

- **Hardware DIFT**: **off-core** [9], off-loading core, in-core

- **Advantage**: no internal hardware modification to the main core.

- **Disadvantage**: needs support from the OS for the synchronization between data and tags.

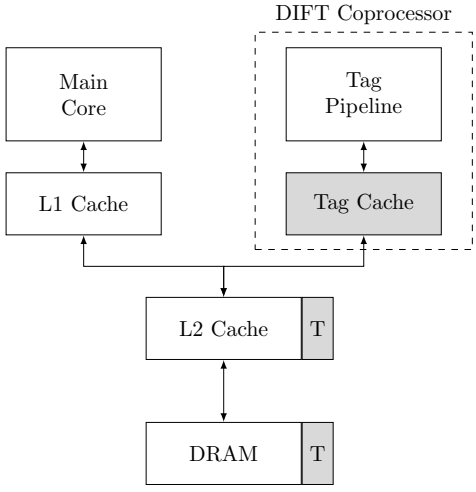


Figure 6: Representation of a Hardware Off-Core DIFT (inspired by [9])

Software threats: Dynamic Information Flow Tracking

- **Hardware DIFT**: off-core, **off-loading core** [10], in-core

- **Advantage**: hardware does not need to know DIFT tags and policies, and no synchronization is needed.

- **Disadvantage**: requires a multicore CPU, reducing the number of cores available and increase the power consumption.

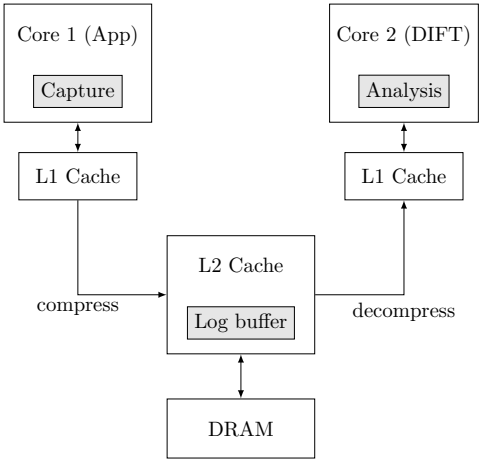


Figure 7: Representation of a Hardware Off-Loading DIFT (inspired by [9])

- **Hardware DIFT**: off-core, off-loading core, **in-core** [11]

- **Advantage**: no multicore CPU and no synchronization are needed. Very low performances overhead.

- **Disadvantage**: highly invasive modifications of internal hardware for tags computations and storing.

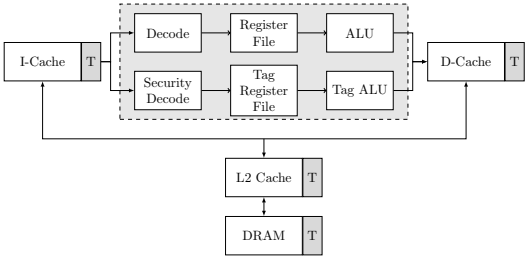


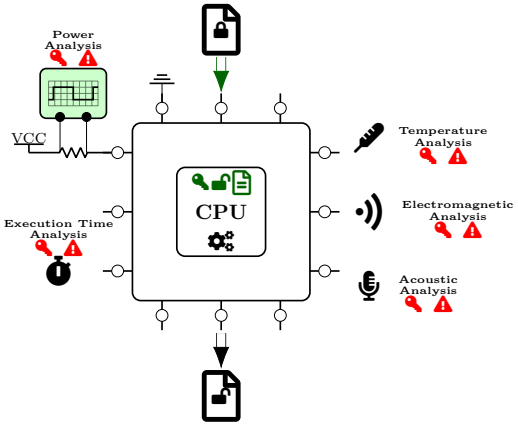
Figure 8: Representation of a Hardware In-Core DIFT (inspired by [9])

Software and Hardware threats

- DIFTs can protect efficiently a system against software attacks
- What would happen if the DIFT were disturbed?
- Considering a tag, what happens if a tag is modified?

Hardware threats: Side-Channel Attacks

- **Side-Channel Attacks** (SCA): involve observing the behaviour of a system to identify potential vulnerabilities.
- Several ways of injecting faults
- Time consumming - lot of traces required
- TEMPEST program by the NSA in 1972
- Indirect example: distinguish a submarine or a warship from another using sonar
- Funny example: Listening the sound made by a keyboard to recover what is written by a doctor during an appointement [12, 13]



Hardware threats: Fault Injection Attacks

- **Fault Injection Attacks** (FIA): involve introducing on purpose one or more fault(s) into a system to disturb its behaviour and identify potential vulnerabilities.
- Several ways of injecting faults
- The precision may vary depending on the category used

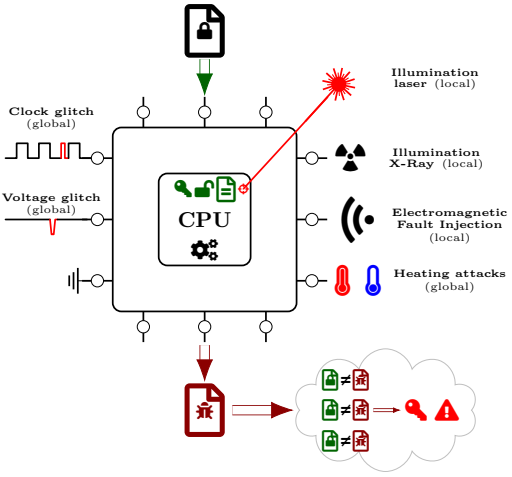
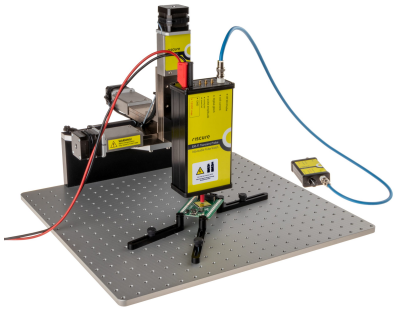


Table 1: Fault Injection methods summary

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

Numerous studies to show vulnerabilities into critical systems

- **Power supply** : manipulations to control the program counter on ARM [14];
- **EM Fault Injection** (EMFI) : to recover an AES key by targeting the cache hierarchy and the MMU [15];
- **Laser Fault Injection** (LFI) : allow the replay of instructions on a 32-bit microcontroller [16].



Numerous studies to show vulnerabilities into critical systems

- **Power supply** : manipulations to control the program counter on ARM [14];
- **EM Fault Injection** (EMFI) : to recover an AES key by targeting the cache hierarchy and the MMU [15];
- **Laser Fault Injection** (LFI) : allow the replay of instructions on a 32-bit microcontroller [16].

▶ No previous studies have shown the vulnerabilities of DIFT against FIA. ◀

Research challenge

How can we maintain maximum protection against software attacks in the presence of physical attacks?

Objectives of this PhD Thesis

- Provide a robust security mechanism against software and hardware threats;
- Propose lightweight countermeasures against FIA;
- Take into account constraints, such as efficiency, area, and performance overhead.

II. D-RI5CY – Vulnerability Assessment

D-RI5CY — origins

- DIFT design [17] made by researchers at Columbia University (USA) with Politecnico di Torino (Italy)
- Based on the 32-bit RISC-V processor: RI5CY (Pulp Platform)
- Open source[1]
- DIFT considering 1-bit tag data path
- Flexible security policy that can be modified at runtime

[1] https://github.com/sld-columbia/riscv-dift

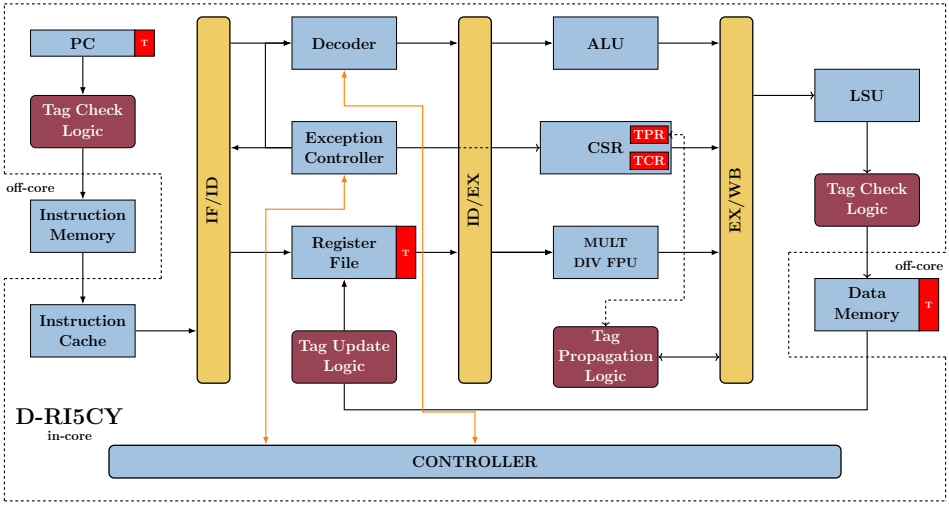


Figure 9: Architecture of the D-RI5CY.

Tag Propagation Register

Table 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 V1	0 0 1	1 0	1 0	0 0	1 0	1 0	0 0	1 0
Policy V2	1 1 1	1 0	1 0	1 0	1 0	1 0	1 0	1 0

- A Mode field for each class of instructions, which specifies how to propagate the tags of the input operands to the output operand tag.
  - the output tag keeps its old value (00);
  - the output tag is set to one, if both the input tags are set to one (01);
  - the output tag is set to one, if at least one input tag is set to one (10);
  - the output tag is set to zero (11).
- The three bits in the L/S enable field allow the policy to enable the source, source-address, and destination-address tags, respectively

Tag Check Register

Table 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 V1	1	1 0 1 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0	0 0 0
Policy V2	0	0 0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0	0 1 1

- The tag-check rules restrict the operations that may be performed on tagged data. If the check bit for an operand tag is set to one and the corresponding tag is equal to one, an exception is raised.
  - For all the classes except Load/Store, there are three tags to consider: first input, second input, and output tags
  - For the Load/Store class there are four tags to take into account: source-address, source, destination-address, and destination tags
  - the additional Execute Check field is associated with the program counter and specifies whether to raise a security exception when the program-counter tag is set to one

Vulnerability Assessment — Why?

We do a vulnerability assessment in order to:

- check if this DIFT is vulnerable against FIA,
- determine the spatial and temporal locations of vulnerabilities.

- Presented at Sensors S&P 2023 [18].

Threat model

We consider an attacker able to:

- perform a physical attack to defeat the DIFT mechanism and realise a software attack,
- inject faults in DIFT-related registers:
  - bit set,
  - bit reset,
  - bit-flip.

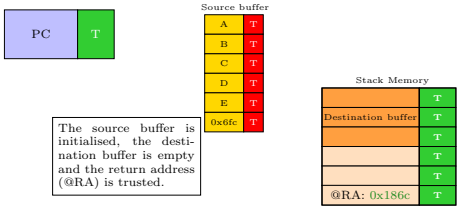
  } Fault model at bit level

Methodology

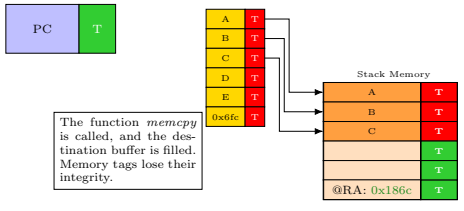
- Analysis of 3 use cases: buffer overflow attack, format string attack, and compare/compute
- We do a temporal, and logical analysis of the tag propagation

Case 1: Buffer overflow

- The attacker exploits a buffer overflow to access the return address register (*RA*).



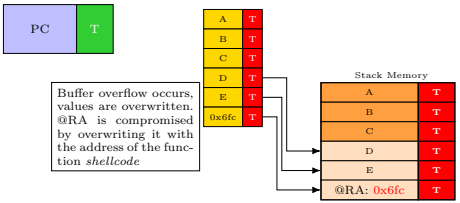
(a) Initialisation



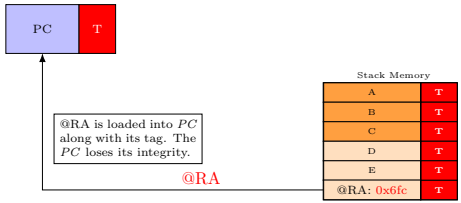
(b) Copy of the source buffer into the destination buffer

- As the data in the source buffer is manipulated by the user, it is marked as *untrusted*.
- Thanks to the DIFT, the tags associated with the source buffer data overwrite the memory tags.

Case 1: Buffer overflow



(a) An overflow occurs, the *RA* register is overwritten



(b) Corrupted *RA* register is loaded into the PC

- Thanks to the DIFT, the tags associated with the source buffer data overwrite the *RA* register tag.
- When the function ends, the corrupted register *RA* is loaded into *PC* using a jalr instruction.

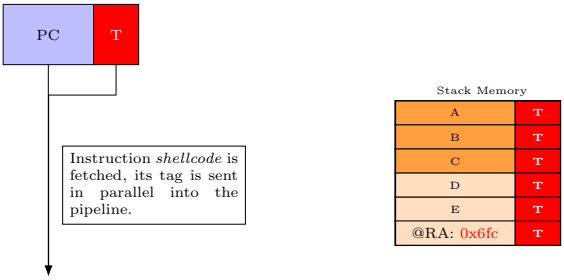


Figure 12: PC address instruction is fetched

- The *PC* has been overwritten, it is now untrusted.
- The *PC* address is fetched to access the next address.

Case 1: Buffer overflow – Temporal analysis of the tag propagation

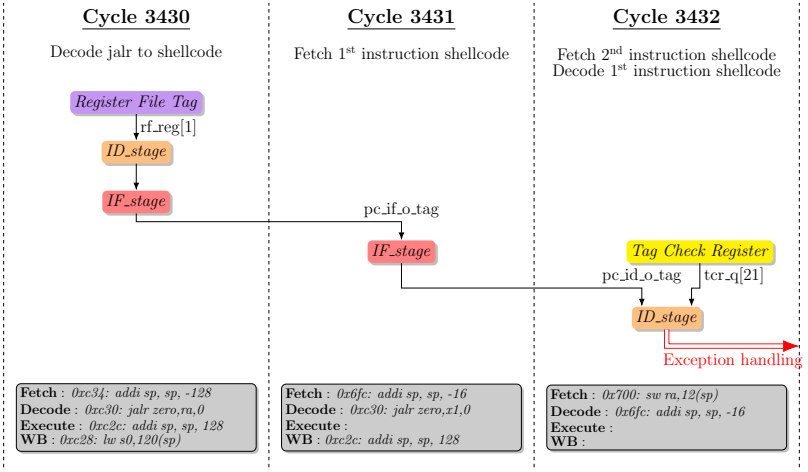


Figure 13: Temporal analysis of tags propagation in a *Buffer Overflow* attack

Case 1: Buffer overflow – Logical analysis of the tag propagation

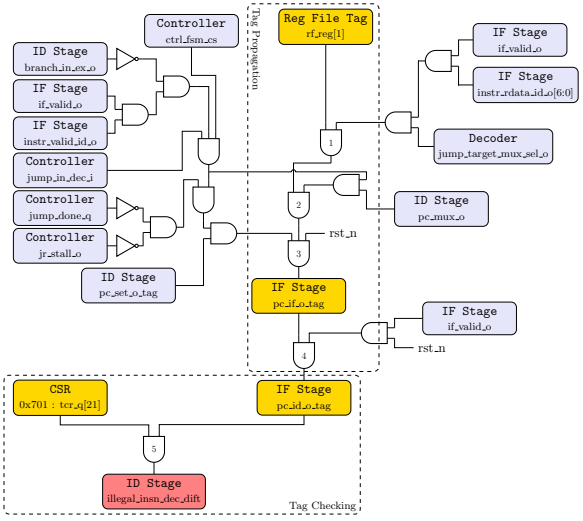


Figure 14: Logical analysis of tags propagation in a *Buffer Overflow* attack

Case 2: WU-FTPd

- The vulnerability is the use of an unchecked user input as the format string parameter in functions that perform formatting, e.g. printf()
- An attacker can use the format tokens, to write into arbitrary locations of memory, e.g. the return address of the function.

```
void echo(){
    int a;
    register int i asm("x8");
    a = i;
    printf("%224u%n%35u%n%253u%n%n", 1, (int*) (a-4), 1, (int*) (a-3), 1, (int*) (a-2), (int*) (a
    -1));
}
```

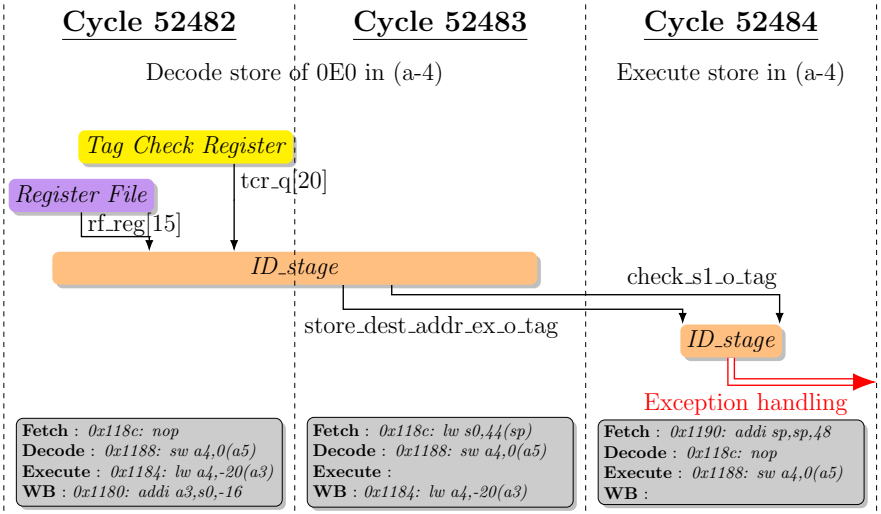


Figure 15: Temporal analysis of the tags propagation in a *format string* attack

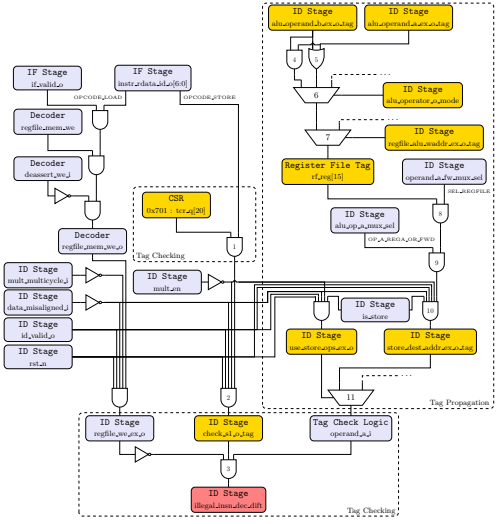


Figure 16: Logical analysis of the tags propagation in a *format string* attack

Experimental Setup — Simulation fault injections campaign

- Logical fault injection simulation is used for preliminary evaluations
  - faults are injected in the HDL code at cycle accurate and bit accurate level
  - a set of 55 DIFT-related registers are targeted
  - a reference simulation is done without fault
  - results are classed in four groups
    - crash: reference cycle count exceeded,
    - silent: current faulted simulation is the same as the reference simulation
    - delay: illegal instruction is delayed
    - success: DIFT has been bypassed
- Simulations with QuestaSim 10.6e.
- FISSA (presented later) is used in order to automate our injection campaigns

Table 4: End of simulation status

	Crash	Silent	Delay	Success	Total
Buffer overflow	0	1380	20	24 (1.69%)	1422
Format string	0	1767	77	52 (2.74%)	1896

Table 5: Buffer overflow : Register sensitivity as determined by fault model and simulation time

	Cycle 3428			Cycle 3429			Cycle 3430			Cycle 3431			Cycle 3432		
	Bit reset	Bit set	Bit flip	Bit reset	Bit set	Bit flip	Bit reset	Bit set	Bit flip	Bit reset	Bit set	Bit flip	Bit reset	Bit set	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]			✓			✓									

Summary

- 4266 simulations have been performed,
- 95 successes (2.23%).
- This campaign showed 43 highly sensitive registers on 55 DIFT-related registers
- We have shown that the D-RI5CY DIFT is vulnerable to FIA
- Propagation of faults is facilitated by paths fully made of *AND* gates

III. Fault Injection Simulation for Security Assessment

Fault Injection Simulation for Security Assessment (FISSA)

Presentation

- Open-Source tool [19].
- Allows the circuit designer to analyse throughout the design cycle the sensibility against FIA.
- Integrated around an HDL Simulator (Questasim).
- The generated results can help to find vulnerabilities during the design phase.
- FISSA enables the principles of Security by Design.
- Presented at DSD 2024 [20].

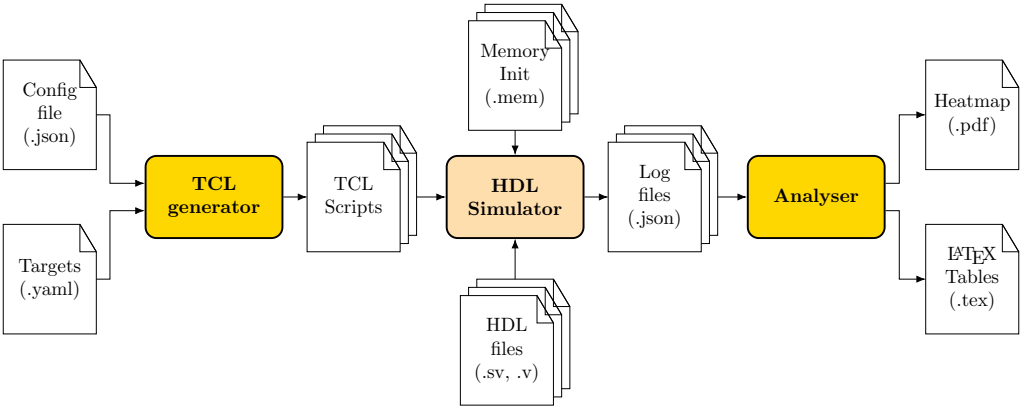


Figure 17: FISSA Software Architecture

FISSA — Supported Fault Models

Table 6: Supported Fault Model

Fault model	Number of target(s)	Number of cycles	Target size
Set to 0	1	1	all
Set to 1	1	1	all
Single bit-flip in one target at a given clock cycle	1	1	all
Single bit-flip in two targets at a given clock cycle	2	1	all
Single bit-flip in two targets at two different clock cycles	2	2	all
Exhaustive multi-bits faults in one target at a given clock cycle	1	1	[1;10] bits
Exhaustive multi-bits faults in two targets at a given clock cycle	2	1	[1;10] bits

- All these fault models are used in this work.

IV. Solutions to Protect against FIAs

Protections

- Focusing on lightweight hardware countermeasures:
  - **Hardware redundancy**: duplication, or triplication, of the circuit to compare the results obtained to check for any difference;
  - **Temporal redundancy**: repeating operations in reverse to compare the result with the initial value;
  - **Instruction replay**: executing multiple times the same instruction or block of instructions;
  - **Obfuscation**: addition of dummy cycles, or shuffle the data;
  - **Information redundancy**: adding additional data to the information to detect or correct the initial value, such as simple parity code, Hamming Code, BCH code, or Reed-Solomon.

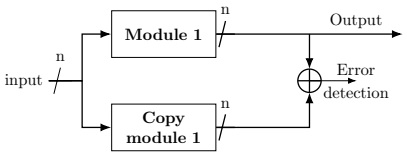


Figure 18: Hardware redundancy

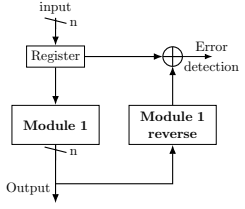


Figure 19: Temporal redundancy

Protections

- Focusing on lightweight hardware countermeasures:
  - **Hardware redundancy**: duplication, or triplication, of the circuit to compare the results obtained to check for any difference;
  - **Temporal redundancy**: repeating operations in reverse to compare the result with the initial value;
  - **Instruction replay**: executing multiple times the same instruction or block of instructions;
  - **Obfuscation**: addition of dummy cycles, or shuffle the data;
  - **Information redundancy**: adding additional data to the information to detect or correct the initial value, such as simple parity code, Hamming Code, BCH code, or Reed-Solomon.

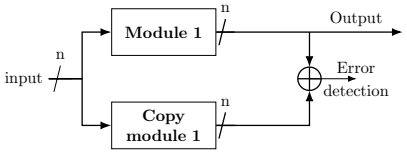


Figure 18: Hardware redundancy

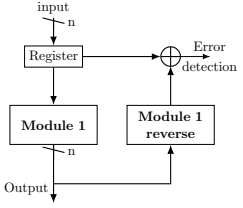


Figure 19: Temporal redundancy

Protections

- Focusing on information redundancy codes for IoT devices:
  - Simple parity
  - Hamming Code
  - Hamming Code with an additional bit (SECDED)
- Only adds a few bits to detect and correct $\Rightarrow$ small overhead on area
- Implementations of *Hamming Code* and *Simple parity* have been presented at ISVLSI 2024 [21].

Detection of single-bit errors — Simple Parity

- Often used for error detection.
- Add an extra bit for parity computation.
- Can only detect one error without correction.

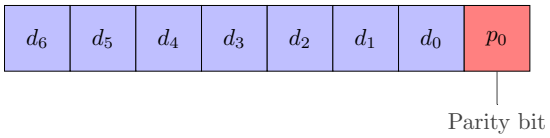


Figure 20: Simple parity codeword

Simple Parity - Example

We consider that we have a value defined as 1001101_2 (77_{10}) in a register protected with single parity.

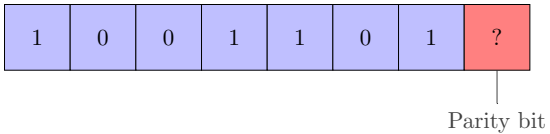


Figure 21: Example with 7 bits of data and 1 parity bit

Simple Parity - Example

We consider that we have a value defined as 1001101_2 (77_{10}) in a register protected with single parity.

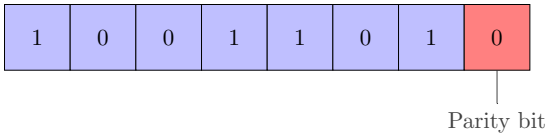


Figure 21: Example with 7 bits of data and 1 parity bit

Simple Parity - Faulted example

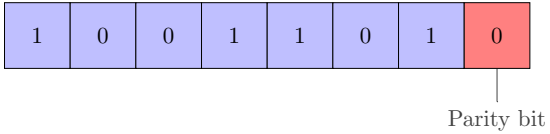


Figure 22: Example with 7 data bits and 1 parity bit

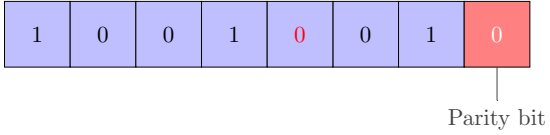


Figure 23: Example where d_2 is faulted

- Linear error-correcting codes, invented by Richard W. Hamming [22].
- Considered as a *perfect code*: the smaller code possible with the same capacity of correction.
- Mostly used in digital communication and data storage systems (ECC memory), satellites.
- Detect and correct single-bit error.
- Redundancy bits are placed in power of 2 positions.
- The number of redundancy bits depends on the number of bits to be protected ($2^r \geq m + r + 1$).

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

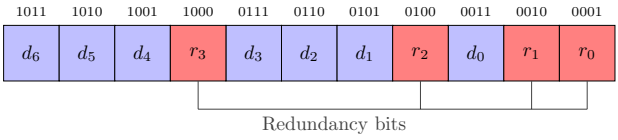


Figure 24: Hamming codeword

Once again, we consider that we have the value 1001101_2 (77_{10}) in a register protected by the Hamming code. The data bits will be placed as shown in Figure 25.

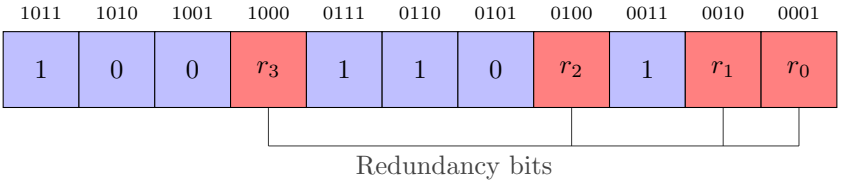
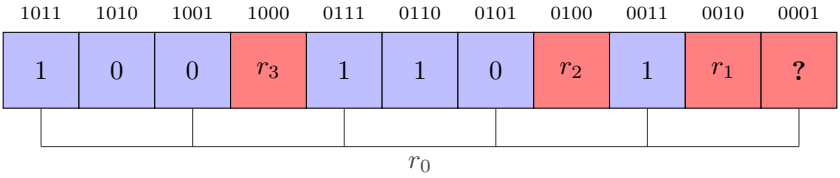
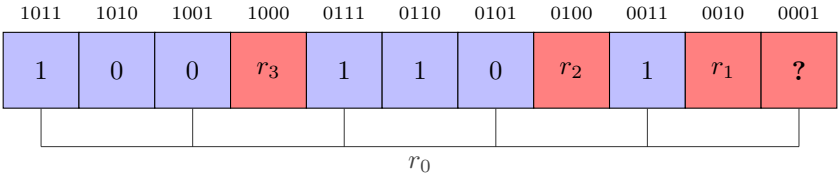


Figure 25: Example with 7 data bits and 4 redundancy bits

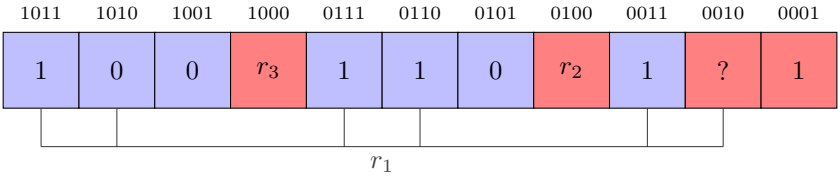
Hamming Code - Example: computation of r_0



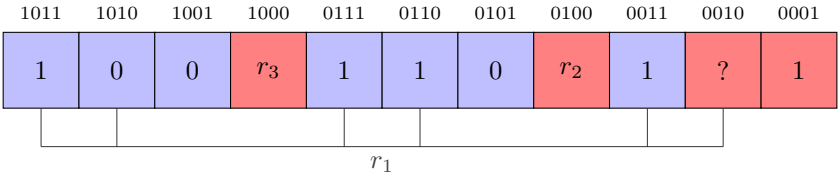
Hamming Code - Example: computation of r_0



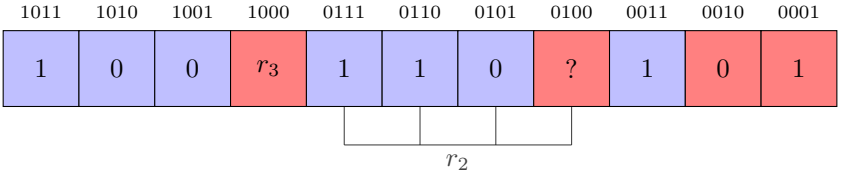
Hamming Code - Example: computation of r_1



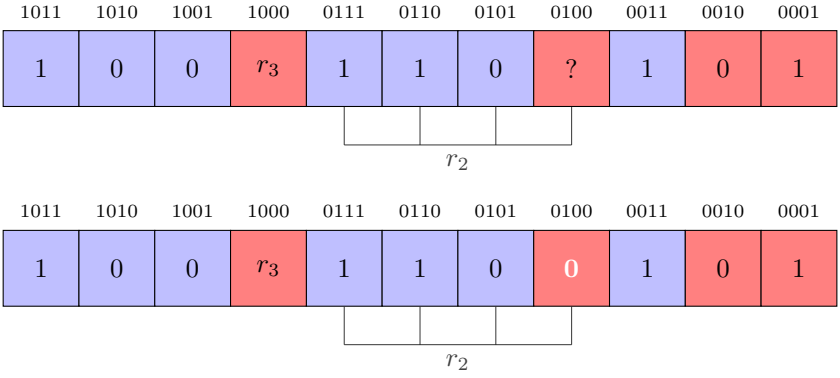
Hamming Code - Example: computation of r_1



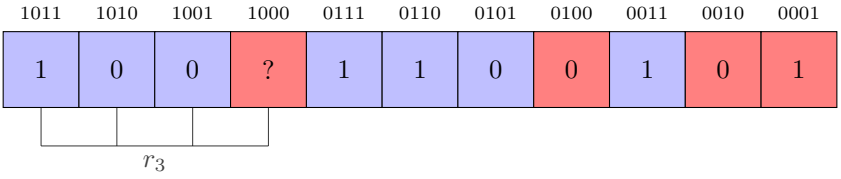
Hamming Code - Example: computation of r_2



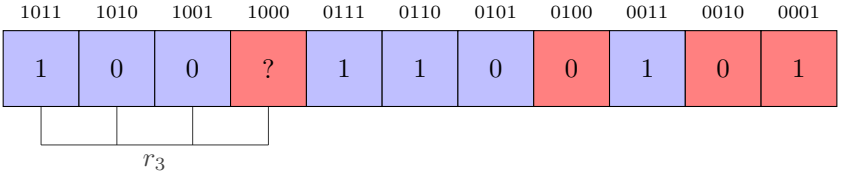
Hamming Code - Example: computation of r_2



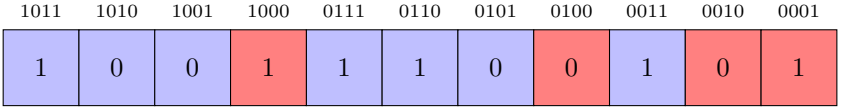
Hamming Code - Example: computation of r_3



Hamming Code - Example: computation of r_3



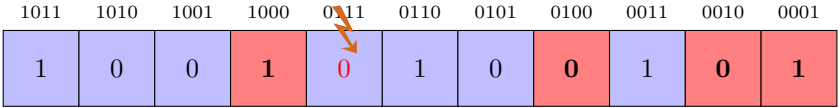
Hamming Code - Example: final register



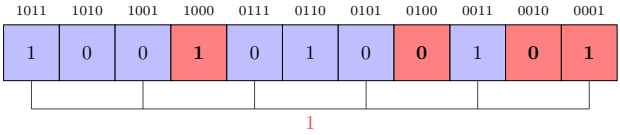
We have now the message 10011100101_2 (1253_{10}).

Hamming Code - Example: bit-flip on d_3

Bit flip on data bit d_3 positioned on the 7th bit of the register.

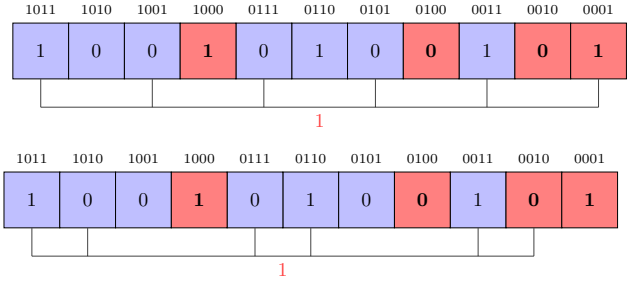


Hamming Code - Example: bit-flip on d_3



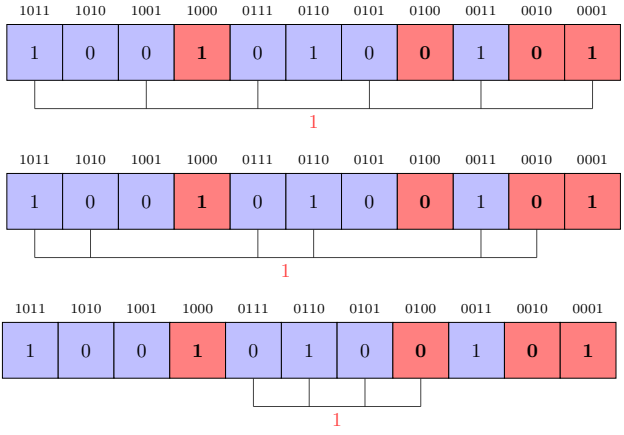
- R0 = 1

Hamming Code - Example: bit-flip on d_3



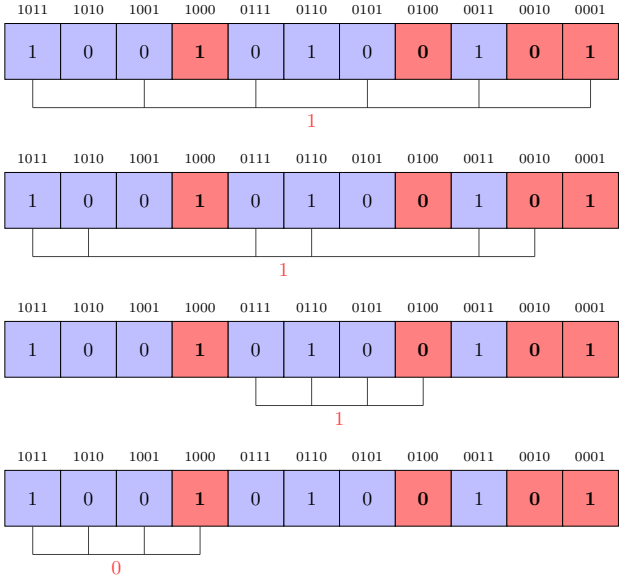
- R0 = 1
- R1 = 1

Hamming Code - Example: bit-flip on d_3



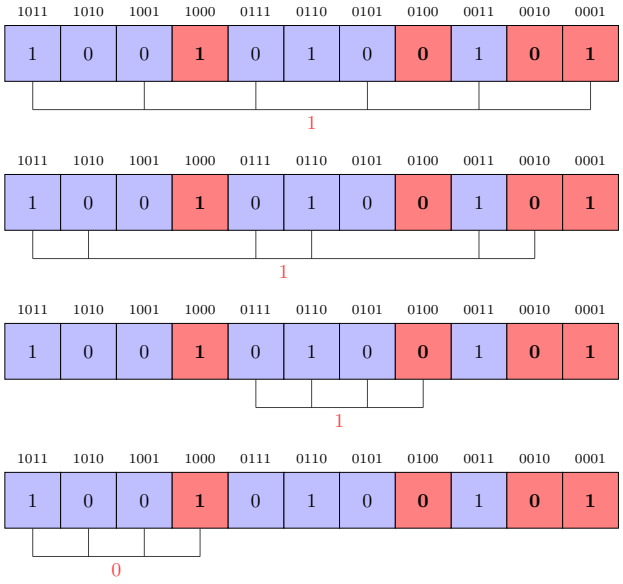
- R0 = 1
- R1 = 1
- R2 = 1

Hamming Code - Example: bit-flip on d_3



- R0 = 1
- R1 = 1
- R2 = 1
- R3 = 0

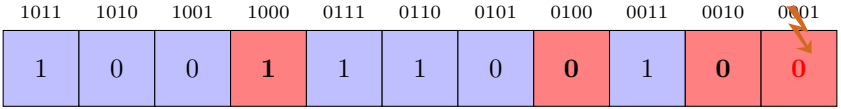
Hamming Code - Example: bit-flip on d_3



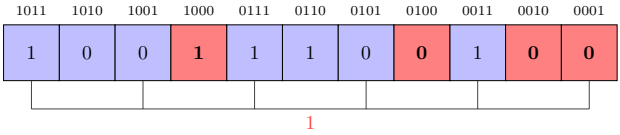
- R0 = 1
- R1 = 1
- R2 = 1
- R3 = 0
- Error = 0111 = 7th bit

Hamming Code - Example: bit-flip on r_0

Bit inversion in the redundancy bit r_0 defined on the first bit of the register.

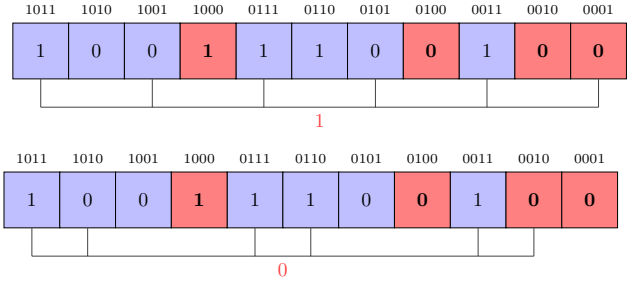


Hamming Code - Example: bit-flip sur r_0



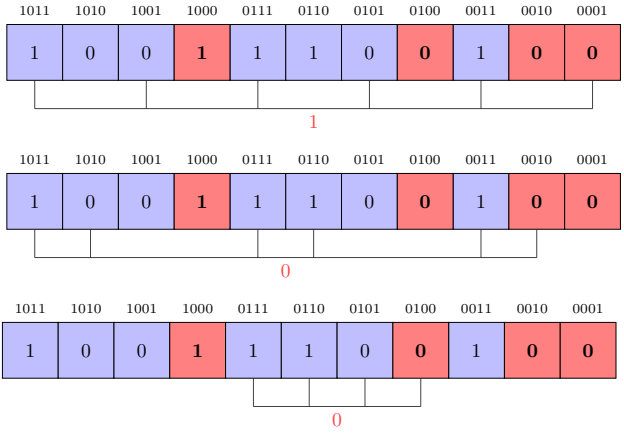
- R0 = 1

Hamming Code - Example: bit-flip sur r_0



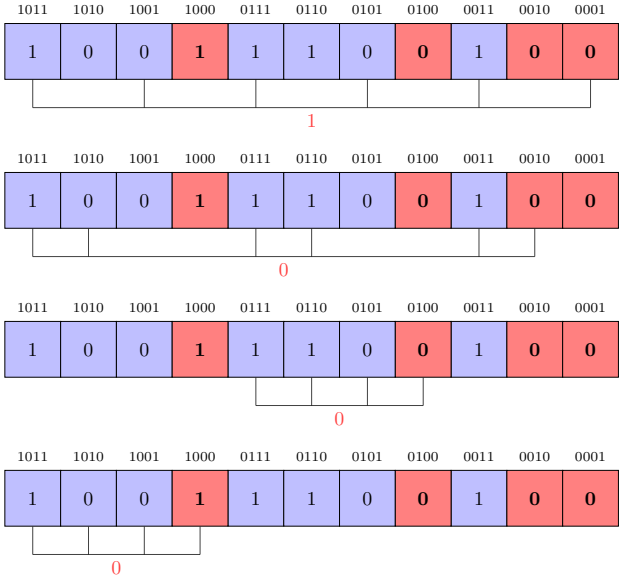
- R0 = 1
- R1 = 0

Hamming Code - Example: bit-flip sur r_0



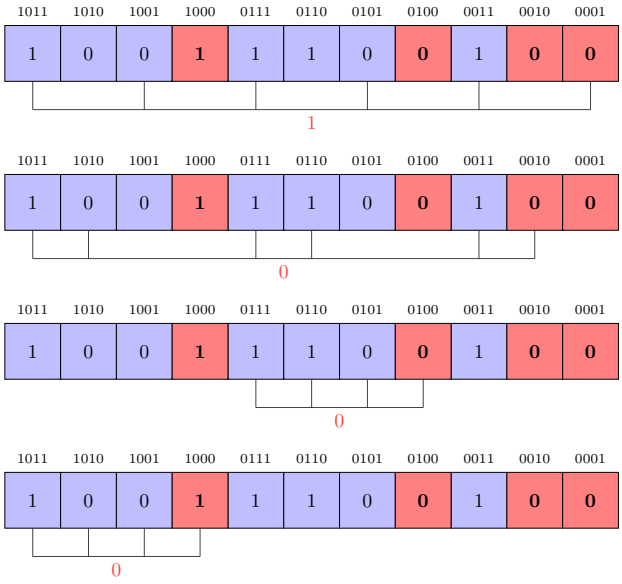
- R0 = 1
- R1 = 0
- R2 = 0

Hamming Code - Example: bit-flip sur r_0



- R0 = 1
- R1 = 0
- R2 = 0
- R3 = 0

Hamming Code - Example: bit-flip sur r_0



- R0 = 1
- R1 = 0
- R2 = 0
- R3 = 0
- Erreur = 0001 = First bit

Single-Error Correction, Double-Error Detection — SECDED

- Based on Hamming Code.
- Detect two-bit error and correct single-bit error.
- An additional bit is added: general parity bit

$$r_0 = d_0 \oplus d_1 \oplus d_3 \oplus d_4 \oplus d_6$$
$$r_1 = d_0 \oplus d_2 \oplus d_3 \oplus d_5 \oplus d_6$$
$$r_2 = d_1 \oplus d_2 \oplus d_3$$
$$r_3 = d_4 \oplus d_5 \oplus d_6$$
$$gp_0 = \bigoplus_{i=0}^{6} d_i \oplus \bigoplus_{j=0}^{3} r_j$$

(2)

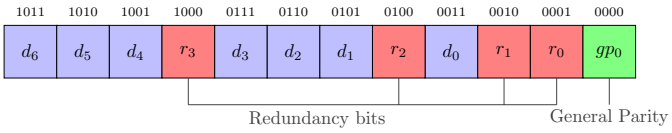


Figure 26: SECDED codeword

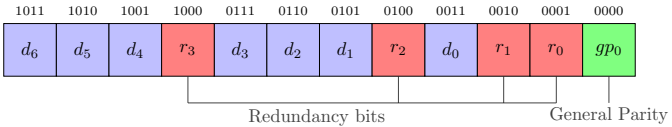


Figure 27: SECDED codeword

Table 7: Summarise of the three case for SECDED

Fault Detection	Redundancy Bits	General Parity Bit
No fault	$\{r_0 - r_3\} = 0$	$gp_0 = 0$
Single Error Correction	$\{r_0 - r_3\} \neq 0$	$gp_0 = 1$
Double Errors Detection	$\{r_0 - r_3\} \neq 0$	$gp_0 = 0$

Once again, we consider that we have the value 1001101_2 (77_{10}) in a register protected by SECDED. The data bits will be placed as shown in Figure 28.

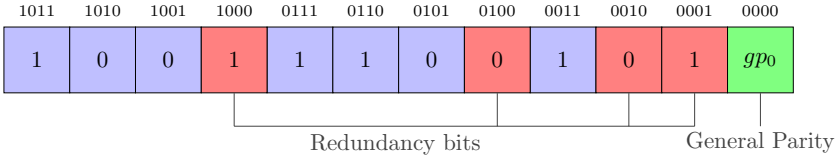
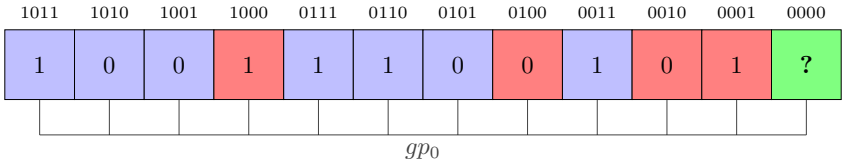
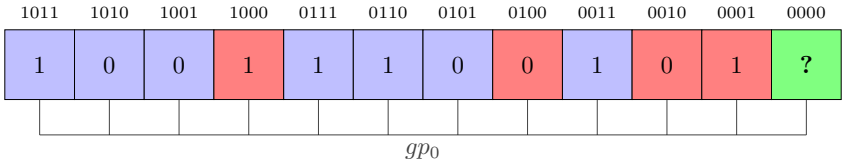


Figure 28: Example with 7 data bits, 4 redundancy bits and 1 parity bit

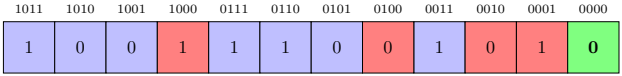
SECDED - Example: computation of gp_0



SECDED - Example: computation of gp_0

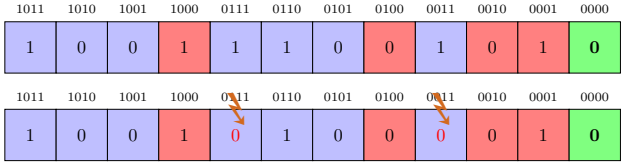


SECDED - Example: 2 bit-flip on d_0 and d_7



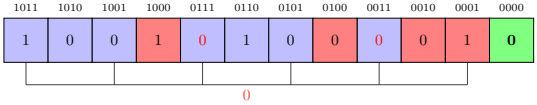
- Initial message.

SECDED - Example: 2 bit-flip on d_0 and d_7



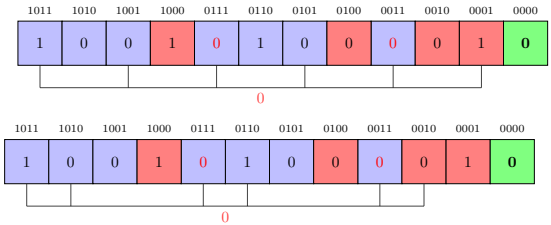
- Initial message.
- Injection of 2 faults in d_0 and d_7.

SECDED - Example: 2 bit-flip on d_0 and d_7



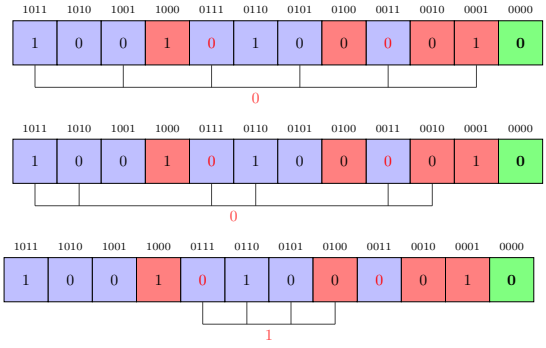
- $R_0 = 0$.

SECDED - Example: 2 bit-flip on d_0 and d_7



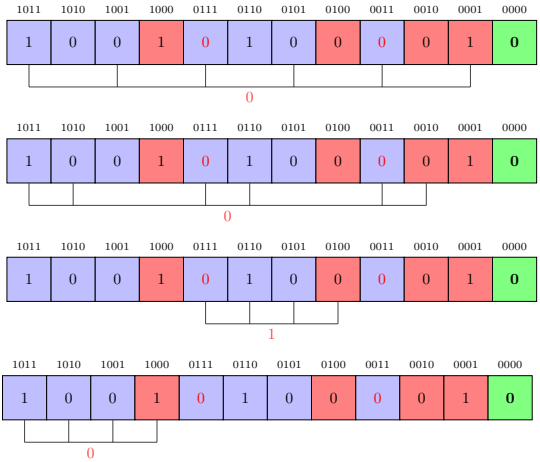
- $R_0 = 0$.
- $R_1 = 0$.

SECDED - Example: 2 bit-flip on d_0 and d_7



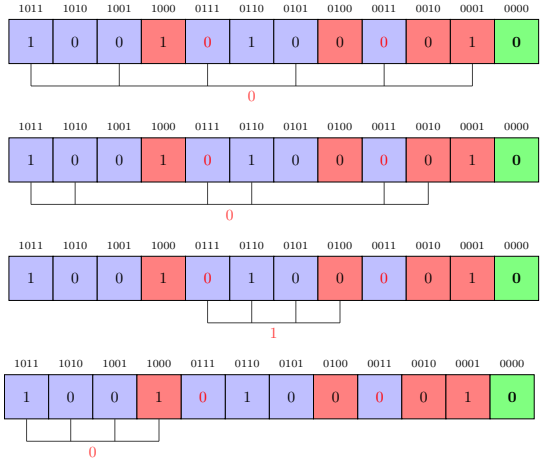
- $R_0 = 0.$
- $R_1 = 0.$
- $R_2 = 1.$

SECDED - Example: 2 bit-flip on d_0 and d_7



- $R_0 = 0$.
- $R_1 = 0$.
- $R_2 = 1$.
- $R_3 = 0$.

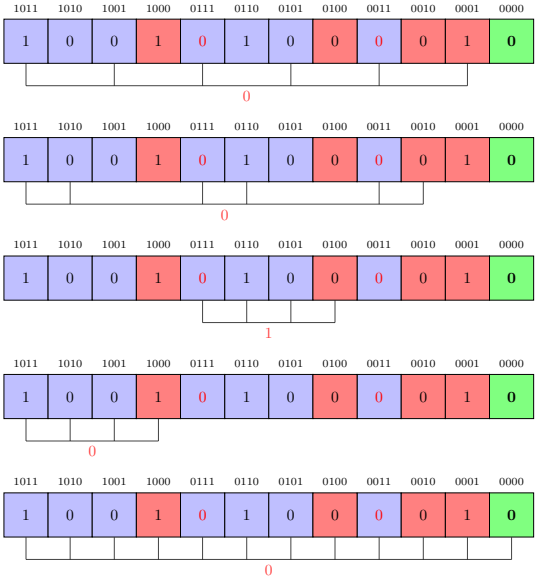
SECDED - Example: 2 bit-flip on d_0 and d_7



- $R_0 = 0$.
- $R_1 = 0$.
- $R_2 = 1$.
- $R_3 = 0$.

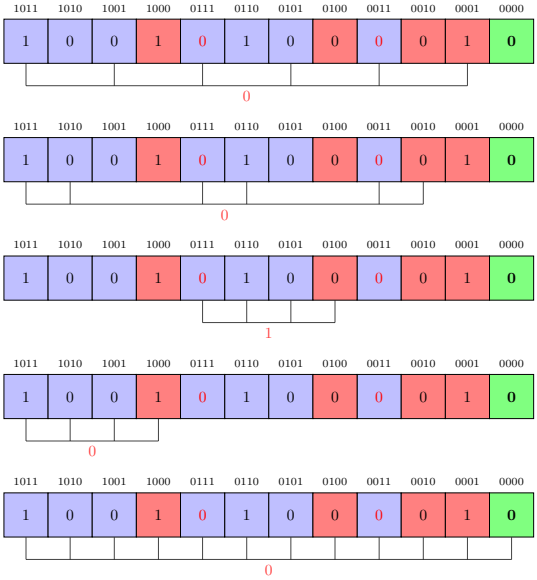
If we consider only the current syndrome 0100_2 with the Hamming code, the code would correct the value at that index (4). It would therefore flip R_2 and set it to 1 even though it was not faulty. This would inject a third fault and further disrupt the processor's behaviour.

SECDED - Example: 2 bit-flip on d_0 and d_7



- $R_0 = 0.$
- $R_1 = 0.$
- $R_2 = 1.$
- $R_3 = 0.$
- $GP_0 = 0.$

SECDED - Example: 2 bit-flip on d_0 and d_7



- $R_0 = 0$.
- $R_1 = 0$.
- $R_2 = 1$.
- $R_3 = 0$.
- $GP_0 = 0$.

We have a syndrome different from 0 and a GP at 0, so it means that it detects a two faults injection and so no correction is done.

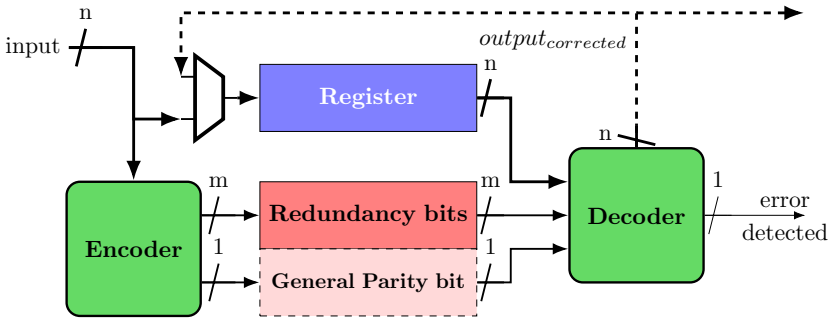


Figure 29: Implementation of a protection for one register

Implementation — Multiple registers for one encoder

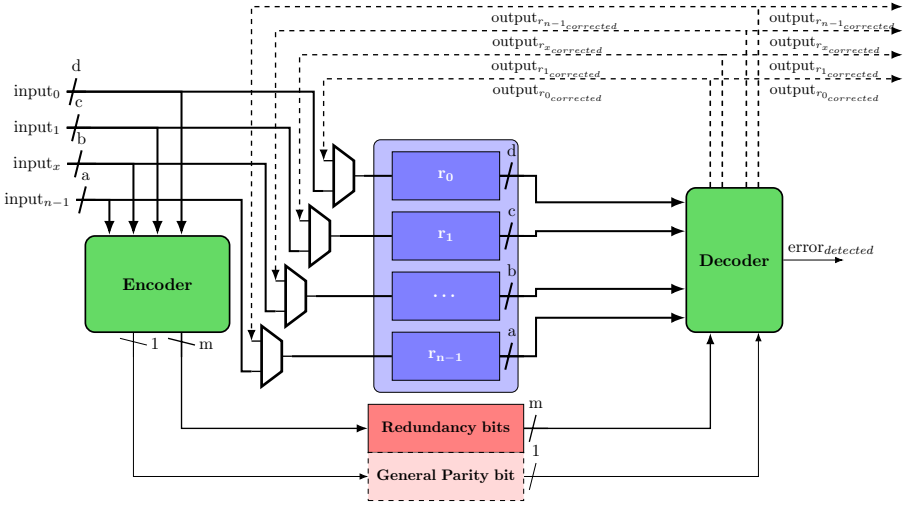


Figure 30: Implementation of a protection for multiple registers

Implementation — Special case for Register File tag

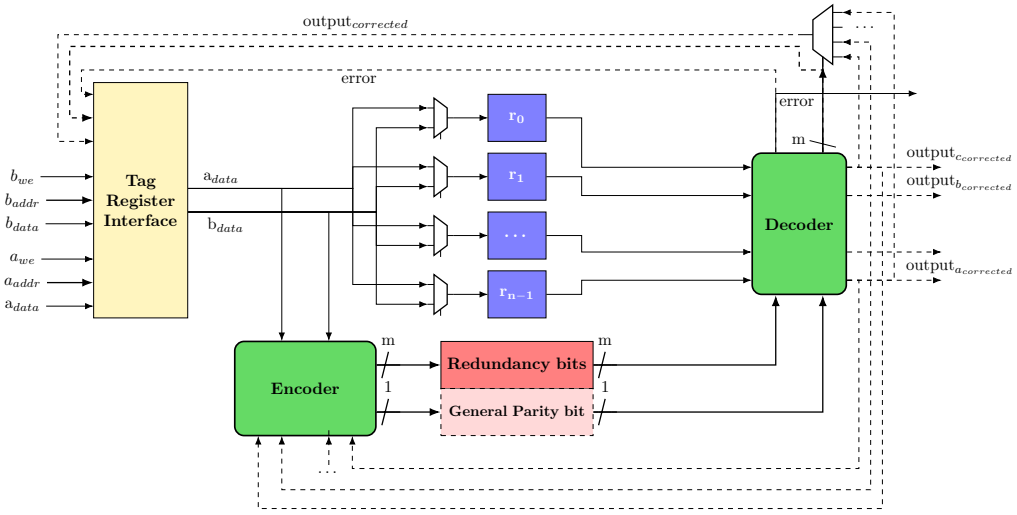


Figure 31: Special implementation for the Register File Tag

Implementation — One register on multiple encoders

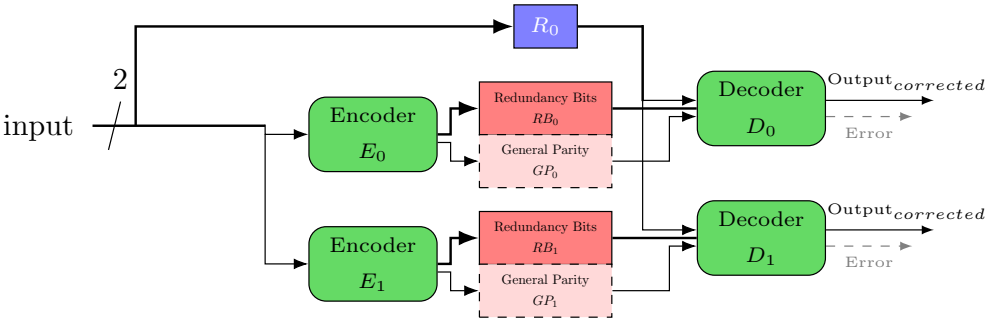


Figure 32: Implementation of a protection for one register split

- Different implementation strategies can be applied depending on protection requirements.
- The protection efficiency would vary
- We want the best protection at the lowest cost possible against different fault models

Table 8: Number of redundancy bit required for a given register size.

Register size	Redundancy bit required
1	1
1	2
2 – 4	3
5 – 11	4
12 – 26	5
27 – 57	6

Table 9: Grouping composition and objectives of implemented strategies

	Grouping strategy	Objective
Strategy 1	Minimisation of groups	Minimisation of the area overhead
Strategy 2	Protection per stage	One protection for each 7 main stages
Strategy 3	Protection per register	Each register is protected individually
Strategy 4	Protection per register with CSR splitting	Strategy 3 + Split the CSRs registers by group of operations
Strategy 5	Coupling split registers	Split each register and couple each bit to another register

Implemented strategies — details

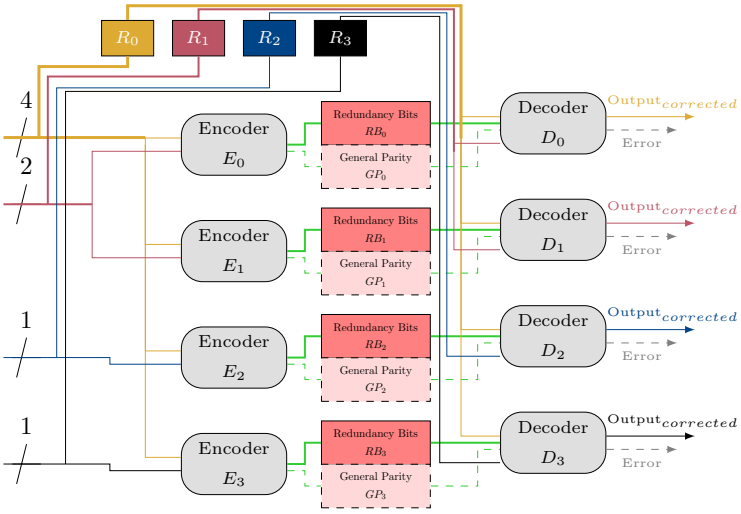


Figure 33: Representation of the fifth strategy

V. Experimental results

Experimental setup

- Use of FISSA for FIA campaigns
- More complex fault models: multi-bit faults or multi single-bit faults

Fault model

- DIFT-related registers + protection-related registers
- Single bit-flip in two registers at two distinct clock cycles ⇒ 1 bit faulted per clock cycle
- Single bit-flip in two registers at a given clock cycle ⇒ 2 bits faulted per clock cycle
- Multi-bit faults in one register at a given clock cycle ⇒ up to 6 bits faulted per clock cycle (registers from 1 to 10 bits only)
- Multi-bit faults in two registers at a given clock cycle ⇒ up to 11 bits faulted per clock cycle (registers from 1 to 10 bits only)

FPGA implementation results

Table 10: FPGA implementation results[2] — Vivado 2023.2

Protection	Number of LUTs	Number of FFs	Maximum frequency
D-RI5CY	6911 (0%)	2335 (0%)	47.6 MHz (0%)
Simple parity	7011 (+1.45%)	2337 (+0.09%)	47.6 MHz (0%)
Hamming Code Strategy 1	7283 (+5.38%)	2361 (+1.11%)	47.4 MHz (-0.36%)
Hamming Code Strategy 2	7369 (+6.63%)	2363 (+1.2%)	46.9 MHz (-1.43%)
Hamming Code Strategy 3	7251 (+4.92%)	2361 (+1.11%)	46.8 MHz (-1.67%)
Hamming Code Strategy 4	7203 (+4.23%)	2371 (+1.54%)	47.6 MHz (0%)
Hamming Code Strategy 5	7182 (+3.92%)	2411 (+3.25%)	47.3 MHz (-0.57%)
SECDED Strategy 1	7428 (+7.48%)	2366 (+1.33%)	47.2 MHz (-0.95%)
SECDED Strategy 2	7433 (+7.55%)	2366 (+1.41%)	47.2 MHz (-0.95%)
SECDED Strategy 3	7324 (+5.98%)	2368 (+1.28%)	47.5 MHz (-0.24%)
SECDED Strategy 4	7255 (+4.98%)	2365 (+1.93%)	48.3 MHz (+1.43%)
SECDED Strategy 5	7228 (+4.59%)	2428 (+3.98%)	48.3 MHz (+1.43%)

[2] Zedboard Xilinx Zynq-7000

FPGA implementation results

Table 10: FPGA implementation results[2] — Vivado 2023.2

Protection	Number of LUTs	Number of FFs	Maximum frequency
D-RI5CY	6911 (0%)	2335 (0%)	47.6 MHz (0%)
Simple parity	7011 (+1.45%)	2337 (+0.09%)	47.6 MHz (0%)
Hamming Code Strategy 1	7283 (+5.38%)	2361 (+1.11%)	47.4 MHz (-0.36%)
Hamming Code Strategy 2	7369 (+6.63%)	2363 (+1.2%)	46.9 MHz (-1.43%)
Hamming Code Strategy 3	7251 (+4.92%)	2361 (+1.11%)	46.8 MHz (-1.67%)
Hamming Code Strategy 4	7203 (+4.23%)	2371 (+1.54%)	47.6 MHz (0%)
Hamming Code Strategy 5	7182 (+3.92%)	2411 (+3.25%)	47.3 MHz (-0.57%)
SECDED Strategy 1	7428 (+7.48%)	2366 (+1.33%)	47.2 MHz (-0.95%)
SECDED Strategy 2	7433 (+7.55%)	2366 (+1.41%)	47.2 MHz (-0.95%)
SECDED Strategy 3	7324 (+5.98%)	2368 (+1.28%)	47.5 MHz (-0.24%)
SECDED Strategy 4	7255 (+4.98%)	2365 (+1.93%)	48.3 MHz (+1.43%)
SECDED Strategy 5	7228 (+4.59%)	2428 (+3.98%)	48.3 MHz (+1.43%)

▶ No major impact on area and performances ◀

[2] Zedboard Xilinx Zynq-7000

Table 11: Logical fault injection simulation campaigns results for single bit-flip in two registers at a given clock cycle

		Crash	Silent	Delay	Detection	Detection & Correction	Double Error Detection	Success	Total	Execution time (h:min)
Buffer Overflow	No protection	0	45 097	1503	–	–	–	1406 (2.93%)	48 006	13:43
	Simple parity	0	10 551	134	40 952	–	–	239 (0.46%)	51 876	14:07
	Hamming 1	0	0	575	–	67 829	–	452 (0.66%)	68 856	19:48
	Hamming 2	0	0	297	–	72 867	–	312 (0.42%)	73 476	97:16
	Hamming 3	0	0	263	–	108 326	–	281 (0.26%)	108 870	30:00
	Hamming 4	0	0	57	–	155 112	–	99 (0.06%)	155 268	46:30
	Hamming 5	0	0	55	–	173 367	–	98 (0.06%)	173 520	53:00
	SECDED 1	0	2436	0	–	59 424	11 616	0	73 476	20:56
	SECDED 2	0	0	0	–	69 354	10 842	0	80 196	21:49
	SECDED 3	0	0	0	–	128 376	9654	0	138 030	40:14
	SECDED 4	0	0	0	–	204 060	7410	0	211 470	64:02
	SECDED 5	0	12 096	0	–	214 722	7542	0	234 360	69:44

Table 12: Logical fault injection simulation campaigns results for exhaustive multi-bits faults in two registers at a given clock cycle

		Crash	Silent	Delay	Detection	Detection & Correction	Double Error Detection	Success	Total	Execution time (h:min)
Buffer Overflow	No protection	0	67 072	926	–	–	–	450 (0.66%)	68 448	11:11
	Simple parity	0	24 622	8	53 359	–	–	59 (0.08%)	78 048	25:00
	Hamming 1	0	294 464	6273	–	–	–	3103 (1.02%)	303 840	99:36
	Hamming 2	0	0	3992	–	319 588	–	4356 (1.33%)	327 936	131:12
	Hamming 3	0	0	4557	–	436 187	–	4408 (0.99%)	445 152	121:20
	Hamming 4	0	0	5446	–	590 953	–	5329 (0.89%)	601 728	167:00
	Hamming 5	0	0	5987	–	714 873	–	5860 (0.81%)	726 720	210:31
	SECDED 1	0	0	1911	–	150 791	170 575	723 (0.22%)	324 000	86:59
	SECDED 2	0	0	1186	–	170 805	184 761	584 (0.16%)	357 336	94:04
	SECDED 3	0	0	1230	–	300 260	263 665	669 (0.12%)	565 824	161:30
	SECDED 4	0	0	18	–	457 498	368 959	61 (0.0074%)	826 536	244:48
	SECDED 5	0	0	39	–	576 992	401 407	66 (0.0067%)	978 504	284:45

Table 13: Logical fault injection simulation campaigns results for single bit-flip in two registers at a given clock cycle

		Crash	Silent	Delay	Detection	Detection & Correction	Double Error Detection	Success	Total	Execution time (h:min)
Format String	No protection	0	55 589	5035	–	–	–	3384 (5.29%)	64 008	163:09
	Simple parity	0	13 361	450	54 590	–	–	767 (1.11%)	69 168	114:06
	Hamming 1	0	0	1709	–	89 010	–	1089 (1.19%)	91 808	179:38
	Hamming 2	0	0	982	–	96 182	–	804 (0.82%)	97 968	136:40
	Hamming 3	0	0	659	–	143 883	–	618 (0.43%)	145 160	261:40
	Hamming 4	0	0	379	–	206 423	–	222 (0.11%)	207 024	368:10
	Hamming 5	0	0	391	–	230 758	–	211 (0.09%)	231 360	445:58
	SECDED 1	0	0	0	–	82 480	15 488	0	97 968	233:28
	SECDED 2	0	0	0	–	92 472	14 456	0	106 928	185:35
	SECDED 3	0	0	0	–	171 168	12 872	0	184 040	317:20
	SECDED 4	0	0	0	–	272 080	9880	0	281 960	462:58
	SECDED 5	0	16 128	0	–	286 296	10 056	0	312 480	558:16

Table 14: Logical fault injection simulation campaigns results for exhaustive multi-bits faults in two registers at a given clock cycle

		Crash	Silent	Delay	Detection	Detection & Correction	Double Error Detection	Success	Total	Execution time (h:min)
Format String	No protection	0	84 419	4836	–	–	–	2009 (2.20%)	91 264	104:15
	Simple parity	0	32 275	147	71 198	–	–	444 (0.43%)	104 064	138:40
	Hamming 1	0	0	20 050	–	375 836	–	9234 (2.28%)	405 120	902:08
	Hamming 2	0	0	17 597	–	408 894	–	10 757 (2.46%)	437 248	774:40
	Hamming 3	0	0	17 926	–	564 154	–	11 456 (1.93%)	593 536	1021:50
	Hamming 4	0	0	20 986	–	767 604	–	13 714 (1.71%)	802 304	1418:24
	Hamming 5	0	0	20 547	–	934 077	–	14 336 (1.48%)	968 960	1690:05
	SECDED 1	0	0	5408	–	194 766	227 655	4171 (0.97%)	432 000	740:21
	SECDED 2	0	0	3611	–	220 568	247 704	4565 (0.96%)	476 448	836:41
	SECDED 3	0	0	3088	–	395 487	351 553	4304 (0.57%)	754 432	1305:36
	SECDED 4	0	0	1939	–	604 649	491 945	3515 (0.32%)	1 102 048	1915:20
	SECDED 5	0	0	1938	–	766 527	535 209	998 (0.08%)	1 304 672	2287:38

Generated heatmaps from FISSA with the current fault model

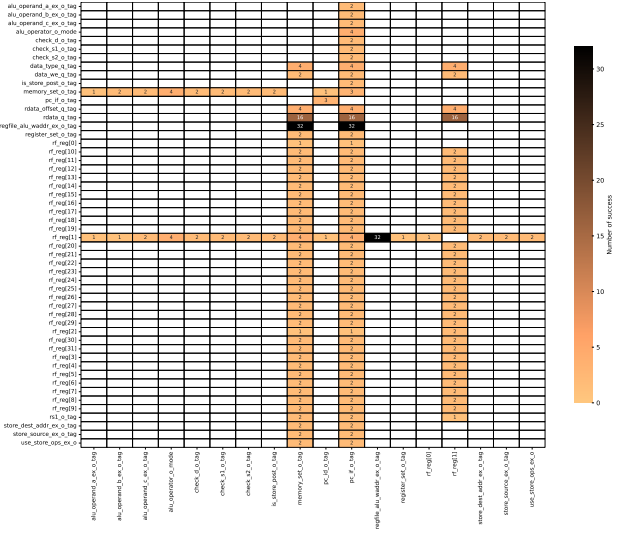


Figure 34: Unprotected version: multi-bits faults in two registers at a given clock cycle → 450 successes

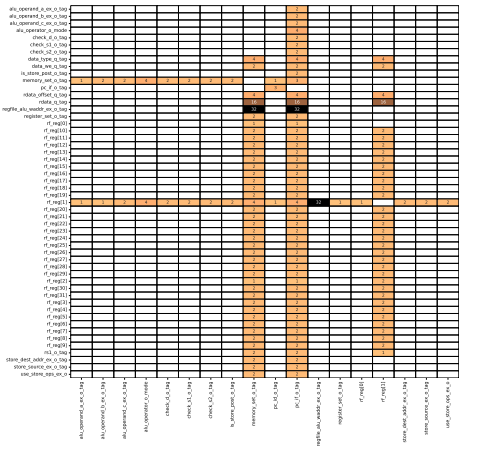


Figure 35: Unprotected version: multi-bits faults in two registers at a given clock cycle → 450 successes

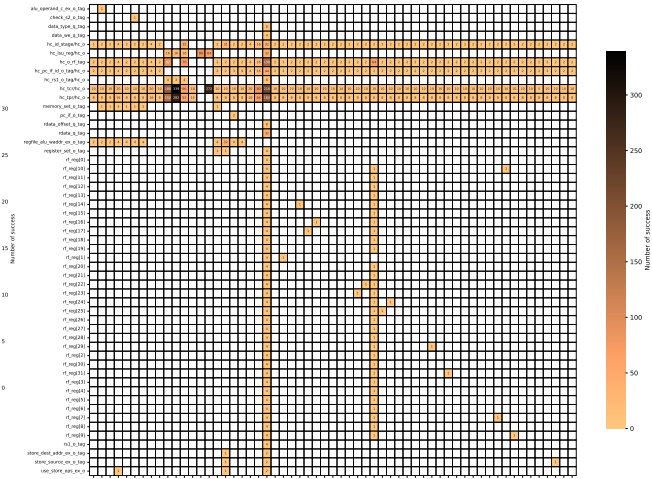


Figure 36: Hamming Code 2 protected version: multi-bits faults in two registers at a given clock cycle → 4356 successes

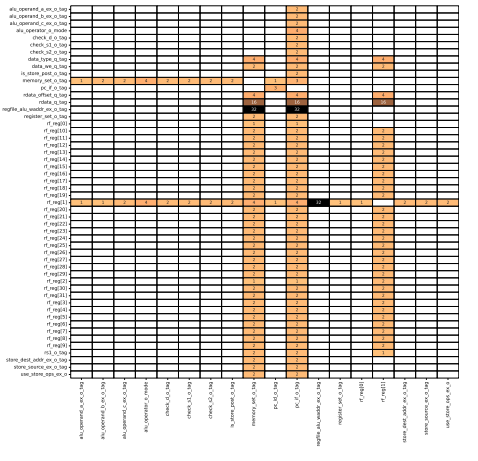


Figure 35: Unprotected version: multi-bits faults in two registers at a given clock cycle → 450 successes

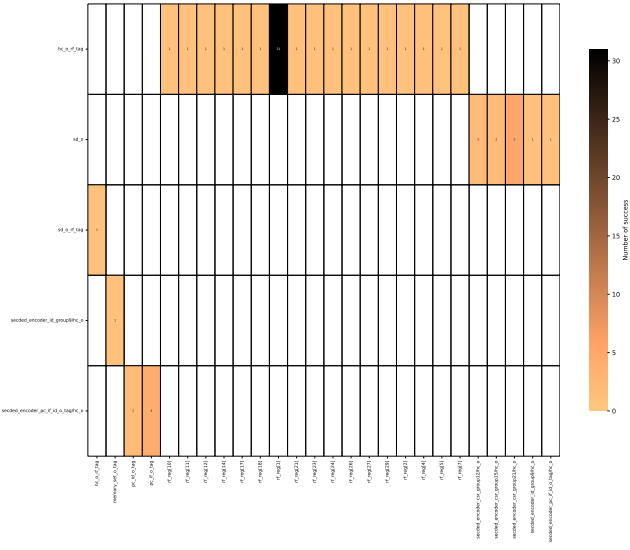


Figure 36: SECDED 5 protected version: multi-bits faults in two registers at a given clock cycle → 66 successes

Part II

VI. Security of AI Implementation

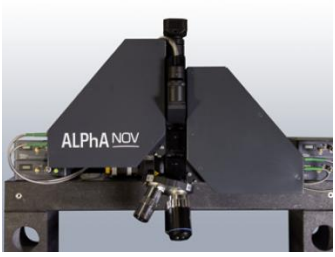
Context and challenges of security of AI implementation

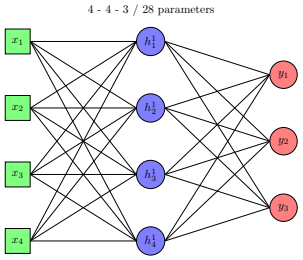
Context

- AI used in many fields, including IoT (edge computing)
- Expensive training of a neural network (GPT-4 $\approx 100M\$$ / *Gemini* $1 \approx 191M\$$).

Objective

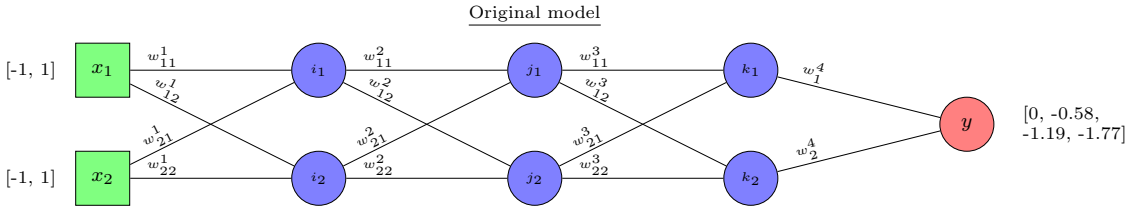
▶ Clone an already trained neural network (MNIST, Iris, ...) using fault injections in flash memory.



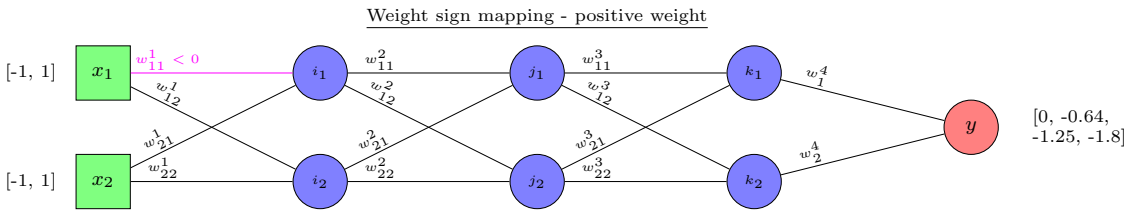


https://github.com/WilliamPsc/Neural-Network-Diagram-Generator

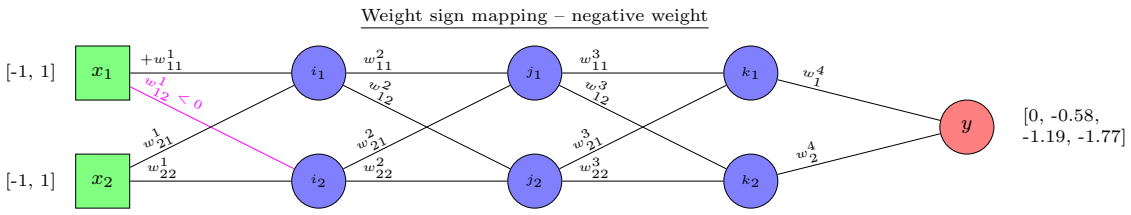
- Original model. Inputs and output values are accessible.



- An error is injected into the MSB of the weight to change the sign. The sign is forced to negative:
  - If there is a change in output, then the weight was positive.

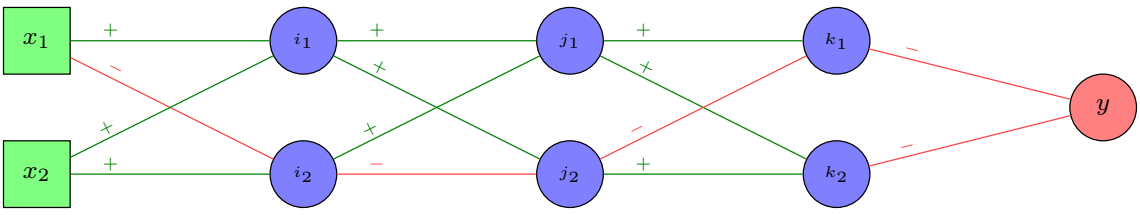


- An error is injected into the MSB of the weight to change the sign. The sign is forced to negative:
  - If there is no change, then the weight was negative.

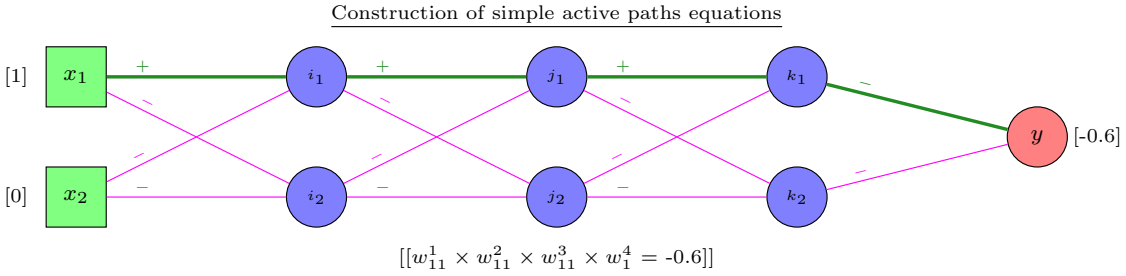


- A complete mapping of the weight signs is obtained.

Complete mapping of weight signs

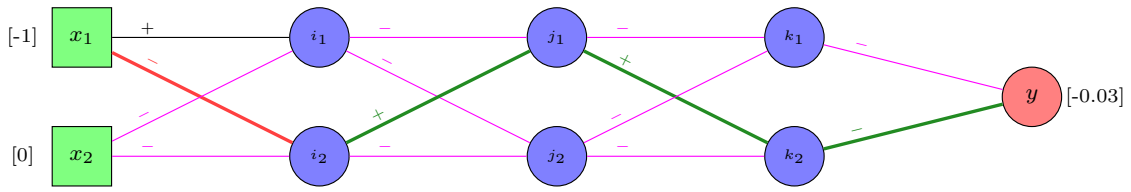


- We construct a system of equations comprising the "*active paths*".



- We construct a system of equations comprising the "*active paths*".

Construction of complex active paths equations



$$[\ [w_{11}^1 \times w_{11}^2 \times w_{11}^3 \times w_1^4 = -0.6],\ [w_{11}^1 \times w_{11}^2 \times w_{12}^3 \times w_2^4 = -0.1],\ [w_{11}^1 \times w_{12}^2 \times w_{22}^3 \times w_2^4 = -0.9],$$
$$[w_{12}^1 \times w_{21}^2 \times w_{11}^3 \times w_1^4 = -0.1],\ [w_{12}^1 \times w_{21}^2 \times w_{12}^3 \times w_2^4 = -0.03]$$
$$[w_{21}^1 \times w_{11}^2 \times w_{11}^3 \times w_1^4 = -0.1],\ [w_{21}^1 \times w_{11}^2 \times w_{12}^3 \times w_2^4 = -0.02],\ [w_{21}^1 \times w_{12}^2 \times w_{22}^3 \times w_2^4 = -0.2],$$
$$[w_{22}^1 \times w_{21}^2 \times w_{11}^3 \times w_1^4 = -0.3],\ [w_{22}^1 \times w_{21}^2 \times w_{12}^3 \times w_2^4 = -0.05]\ ]$$

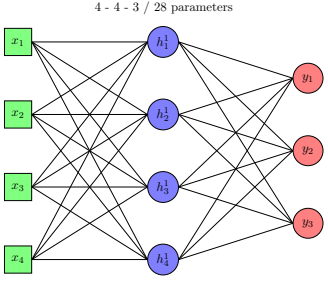
- System resolution using a Python solver that will provide a solution for each of the positive and negative weights contained in the equations.

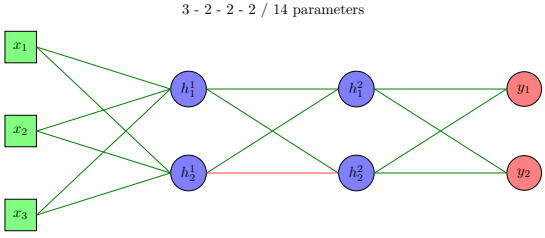
- Compute each of the remaining negative weight values using a second solver.

- Evaluation of the model obtained to verify whether it is equivalent to the original.
  - **Random model:** If it is equivalent, then MSE $==$ 0 or close to it.
  - **Trained model:** If it is equivalent, then the accuracy of the model is identical to that of the original model.

Main results

- Successful cloning on small random models and on models trained with the Iris dataset (MSE at 0 or model accuracy equal to the original).
- It does not matter how many neurons there are on a layer.
- The deeper the network, the more complicated it is.
- Continuation with real experiments using a multispot laser.





Présentation de mes travaux de recherche – Sécurité Matérielle - IA - Contremesures

Présentation CNFM - Mastère SECNUM

William PENSEC

Maître de Conférences
LIRMM – Université de Montpellier
Montpellier, France

Merci pour votre attention.

References

References

[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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] *The 2024 IoT Security Landscape Report*. 2024. URL: https://blogapp.bitdefender.com/hotforsecurity/content/files/2024/06/2024-IoT-Security-Landscape-Report_consumer.pdf.

References

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

References

[15] 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.

[16] Vanthanh Khuat, Jean-Max Dutertre, and Jean-Luc Danger. “Analysis of a Laser-induced Instructions Replay Fault Model in a 32-bit Microcontroller”. In: *24th Euromicro Conference on Digital System Design (DSD)*. 2021, pp. 363–370. DOI: 10.1109/DSD53832.2021.00061.

[17] 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.

[18] 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: ACM, 2023, pp. 8–14. DOI: 10.1145/3628356.3630116.

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

[20] 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)*. Paris, France, Aug. 2024, pp. 369–376. DOI: 10.1109/DSD64264.2024.00056.

[21] 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, pp. 180–185. DOI: 10.1109/ISVLSI61997.2024.00042.

References

[22] 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.

[23] Freepik Company. *Icônes vectorielles*. 2010. URL: https://www.flaticon.com/.