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

PhD Defense

William PENSEC

Université Bretagne Sud, UMR 6285, Lab-STICC, Lorient, France

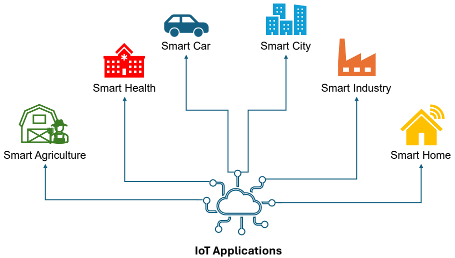
December 19, 2024

Composition of the Jury

Reviewers:	Lejla BATINA
	Vincent BEROULLE
	Nele MENTENS
Examiners:	Jean-Max DUTERTRE
	Francesco REGAZZONI
PhD supervisor:	Guy GOGNIAT
PhD co-director:	Vianney LAPÔTRE

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



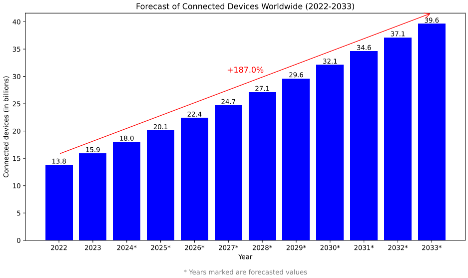


Figure 1: Number of IoT devices worldwide from 2022 to 2033 (from [1])

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]

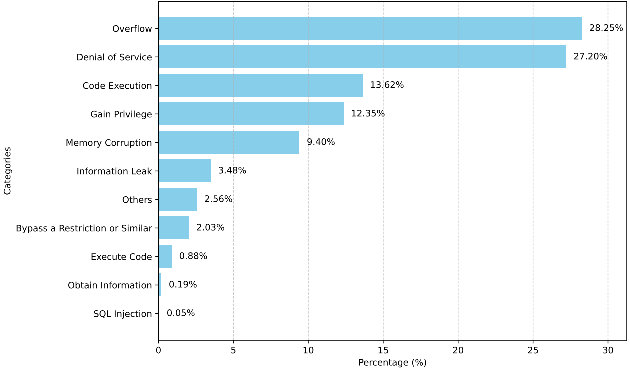


Figure 2: Data from BitDefender [7]

Context: IoT Under Threats

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]

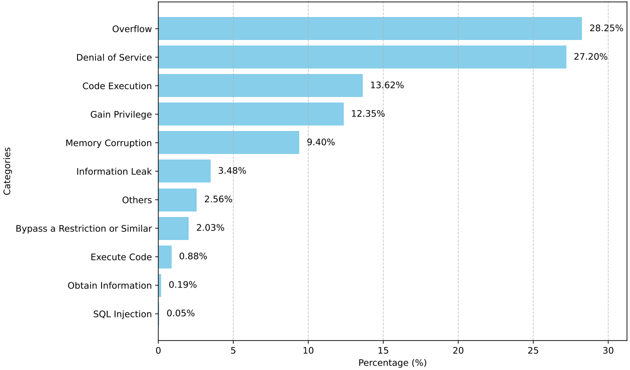


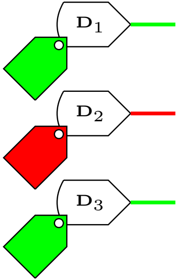
Figure 2: Data from BitDefender [7]

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

Three steps

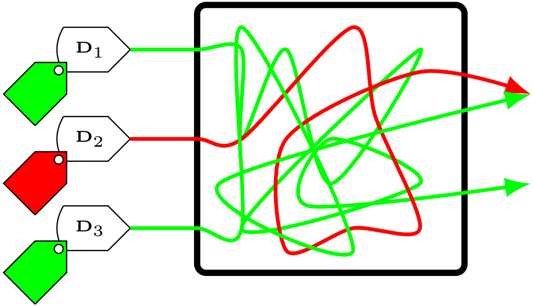
- Tag initialisation
- Tag propagation
- Tag check



Software threats: Dynamic Information Flow Tracking

Three steps

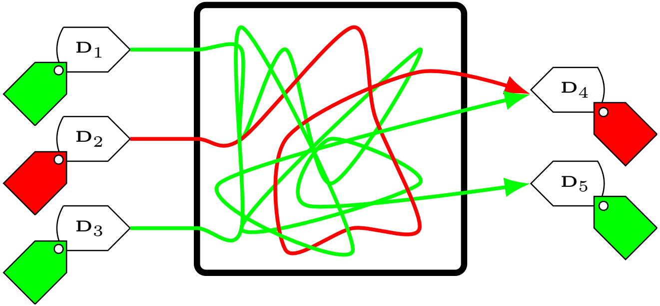
- Tag initialisation
- Tag propagation
- Tag check



Software threats: Dynamic Information Flow Tracking

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.

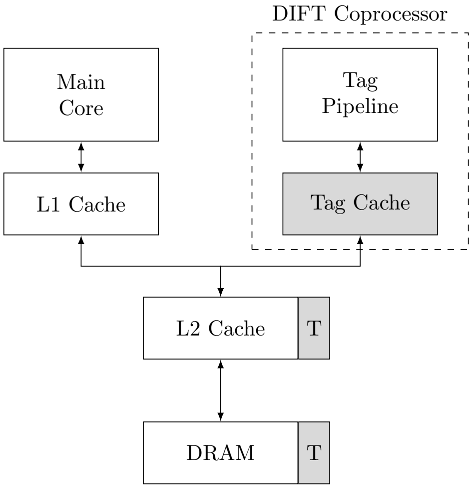


Figure 3: Representation of a Hardware Off-Core DIFT (inspired by [9])

- **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.

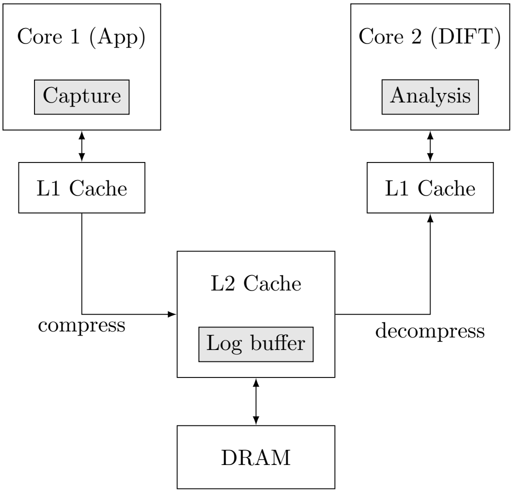


Figure 4: Representation of a Hardware Off-Loading DIFT (inspired by [9])

- **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.

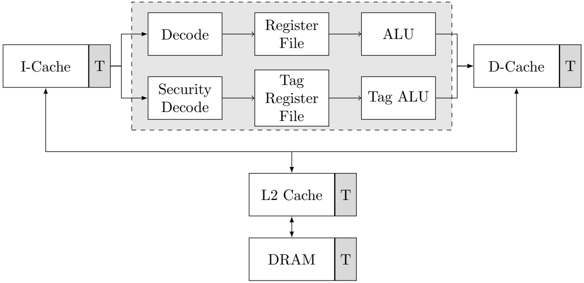


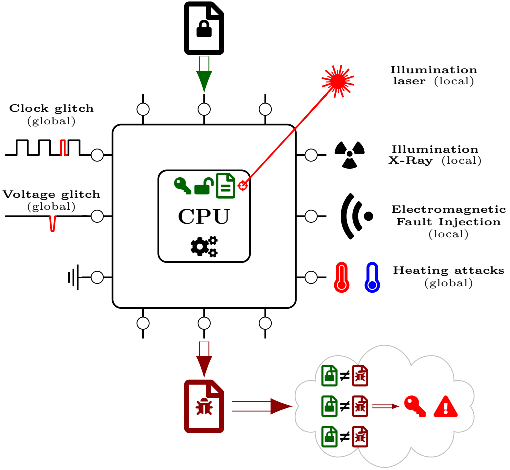
Figure 5: Representation of a Hardware In-Core DIFT (inspired by [9])

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: 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



Motivations

- **Power supply** : manipulations to control the program counter on ARM [12];
- **EM Fault Injection** (EMFI) : to recover an AES key by targeting the cache hierarchy and the MMU [13];
- **Laser Fault Injection** (LFI) : allow the replay of instructions on a 32-bit microcontroller [14].

Motivations

- **Power supply** : manipulations to control the program counter on ARM [12];
- **EM Fault Injection** (EMFI) : to recover an AES key by targeting the cache hierarchy and the MMU [13];
- **Laser Fault Injection** (LFI) : allow the replay of instructions on a 32-bit microcontroller [14].

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

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.

Outline

I. D-RI5CY – Vulnerability Assessment

II. Fault Injection Simulation for Security Assessment

III. Solutions to Protect against FIAs

IV. Experimental results

V. Conclusion and Perspectives

I. D-RI5CY – Vulnerability Assessment

- DIFT design[1] 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[2]
- DIFT considering 1-bit tag data path
- Flexible security policy that can be modified at runtime

[1] 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

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

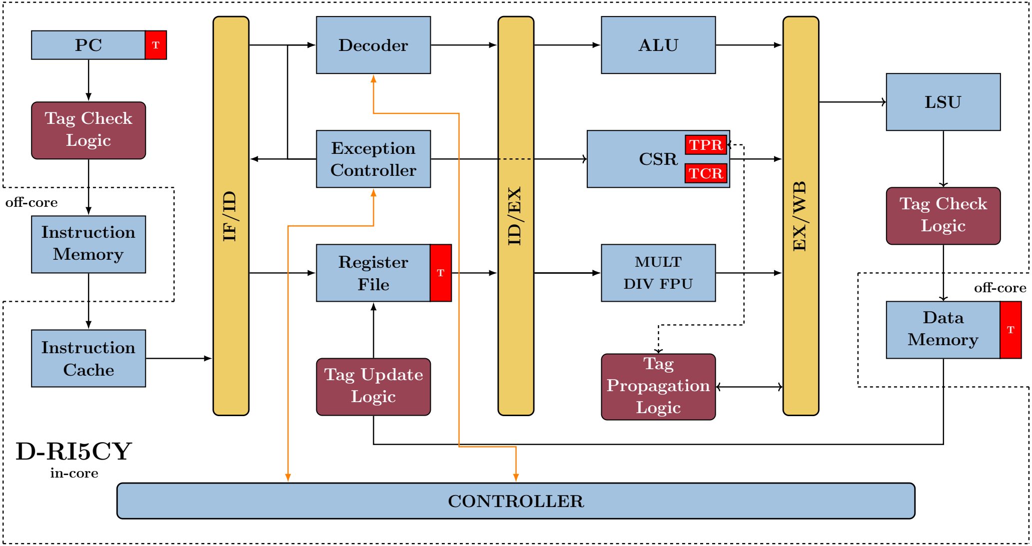


Figure 6: Architecture of the D-RI5CY.

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.

Threat model

We consider an attacker able to:

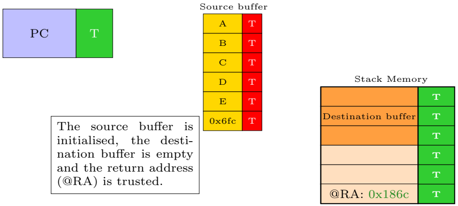
- perform a physical attack to defeat the DIFT mechanism and realise a software attack,
- inject faults in DIFT-related registers:
  - bit set,
  - bit reset,
  - bit-flip.

Fault model at bit level

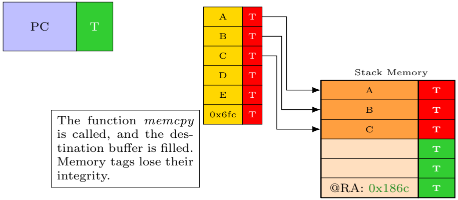
Methodology

- 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

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

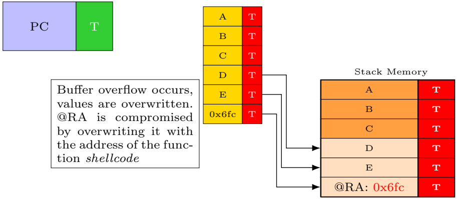


(a) Initialisation

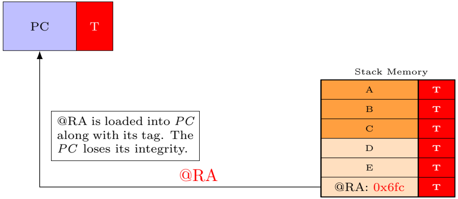


(b) Copy of the source buffer into the destination buffer

- 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.



(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.

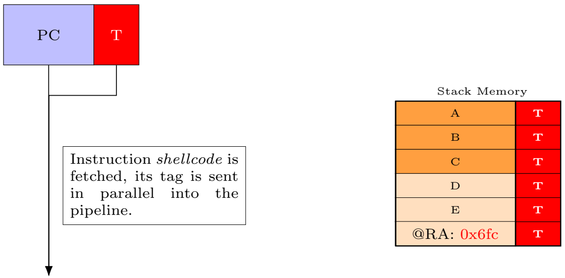


Figure 9: PC address instruction is fetched

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

Temporal analysis of the tag propagation

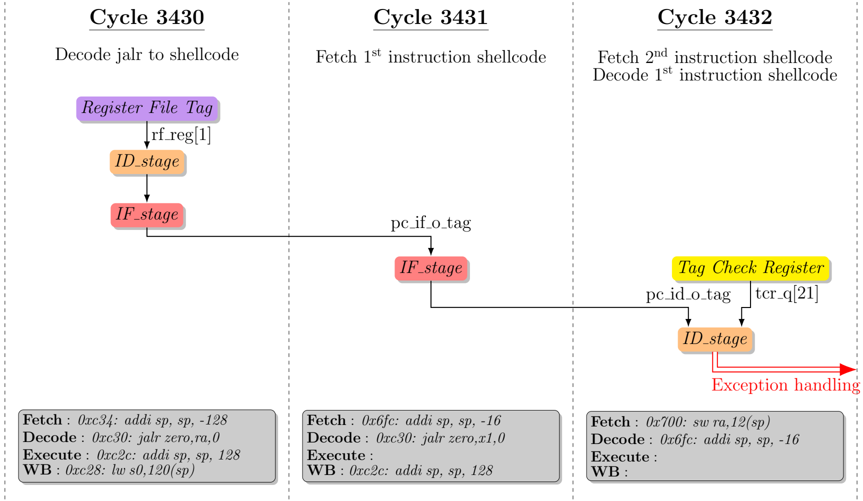


Figure 10: Temporal analysis of tags propagation in a *Buffer Overflow* attack

Logical analysis of the tag propagation

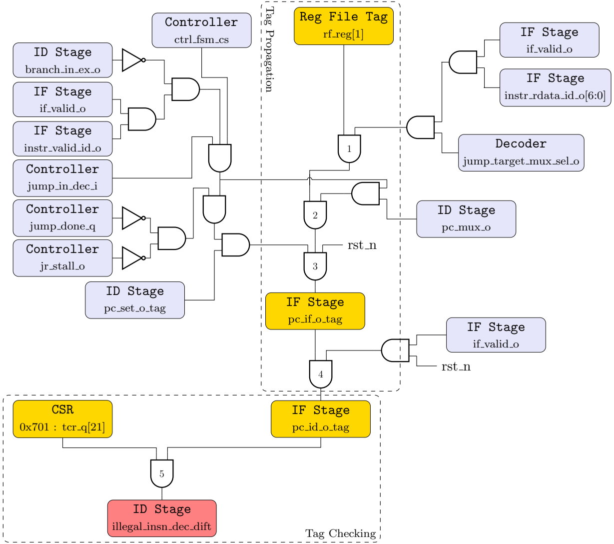


Figure 11: Logical analysis of tags propagation in a *Buffer Overflow* attack

- 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

Buffer overflow

Table 1: End of simulation status

	Crash	NSTR	Delay	Success	Total
Buffer overflow	0	1380	20	24 (1.69%)	1422

Table 2: 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
- Presented at Sensors S&P 2023 [16].

II. Fault Injection Simulation for Security Assessment

Presentation

- Open-Source tool [17].
- 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 [18].

FISSA — Software Architecture

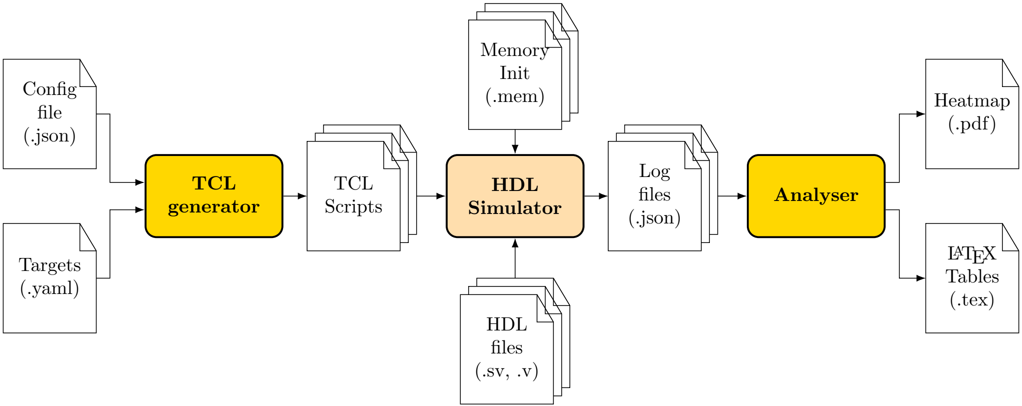


Figure 12: FISSA Software Architecture

Table 3: 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.

III. 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.

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.

Protections

- Focusing on information redundancy codes:
  - Simple parity
  - Hamming Code
  - Hamming Code with an additional bit (SECDED)
- Implementations of *Hamming Code* and *Simple parity* have been presented at ISVLSI 2024 [19].

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.

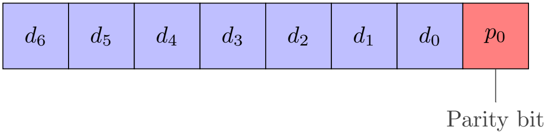


Figure 13: Simple parity codeword

Detection and correction of single-bit errors — Hamming Code

- Linear error-correcting codes, invented by Richard W. Hamming [20].
- Mostly used in digital communication and data storage systems.
- Detect and correct single-bit error.
- Redundancy bits are placed in power of 2 positions.

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

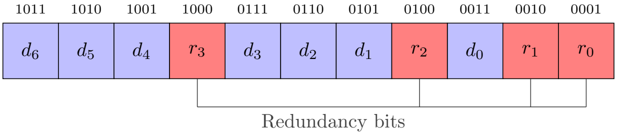


Figure 14: Hamming codeword

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

$$
\begin{aligned}
r_0 &= d_0 \oplus d_1 \oplus d_3 \oplus d_4 \oplus d_6 \\
r_1 &= d_0 \oplus d_2 \oplus d_3 \oplus d_5 \oplus d_6 \\
r_2 &= d_1 \oplus d_2 \oplus d_3 \\
r_3 &= d_4 \oplus d_5 \oplus d_6 \\
gp_0 &= \bigoplus_{i=0}^{6} d_i \oplus \bigoplus_{j=0}^{3} r_j
\end{aligned} \tag{2}
$$

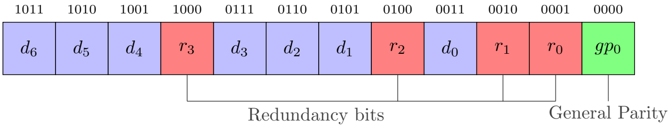


Figure 15: SECDED codeword

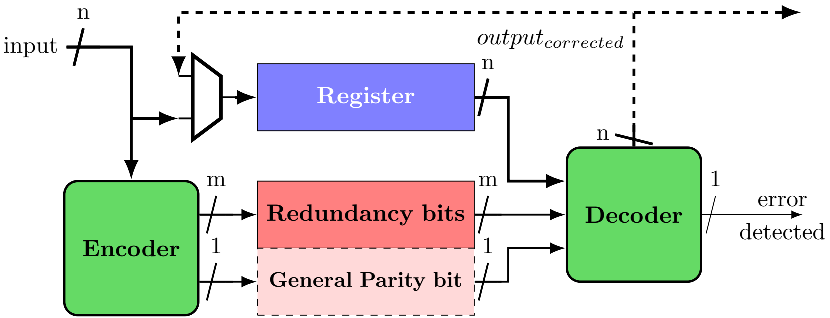


Figure 16: Implementation of a protection for one register

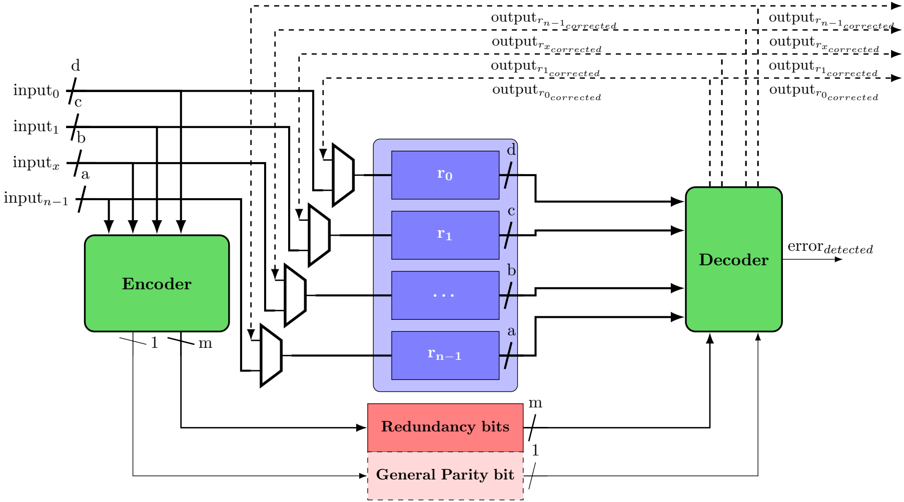


Figure 17: Implementation of a protection for multiple registers

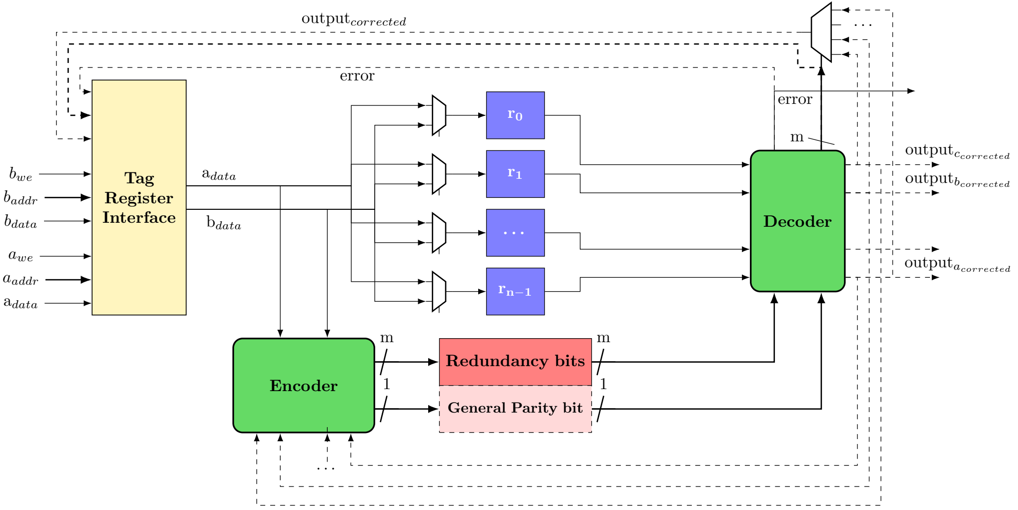


Figure 18: Special implementation for the Register File Tag

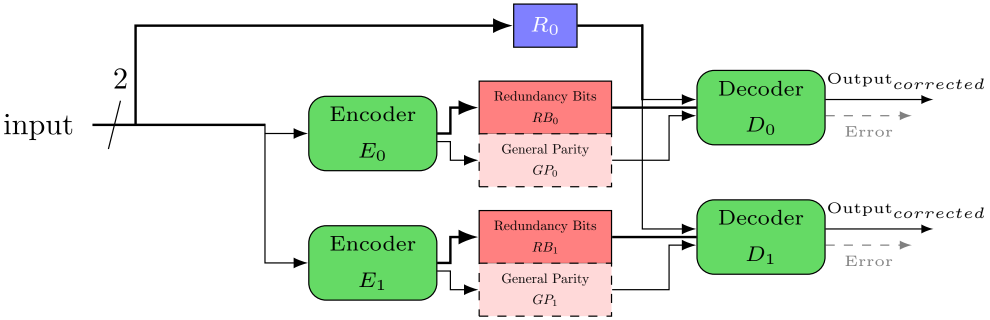


Figure 19: Implementation of a protection for one register split

Implemented strategies — Evaluation of Group Composition

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

Implemented strategies — Group composition

Table 4: 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

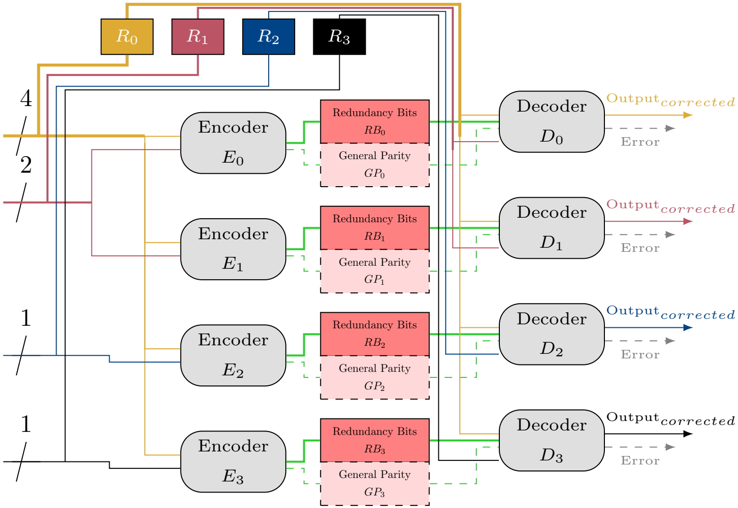


Figure 20: Representation of the fifth strategy

IV. 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 5: FPGA implementation results — 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%)

Table 5: FPGA implementation results — 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 ◀

Obtained results from the first considered fault model

Table 6: 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)
	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
Buffer Overflow	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

Obtained results from the second considered fault model

Table 7: 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

Generated heatmaps from FISSA with the current fault model

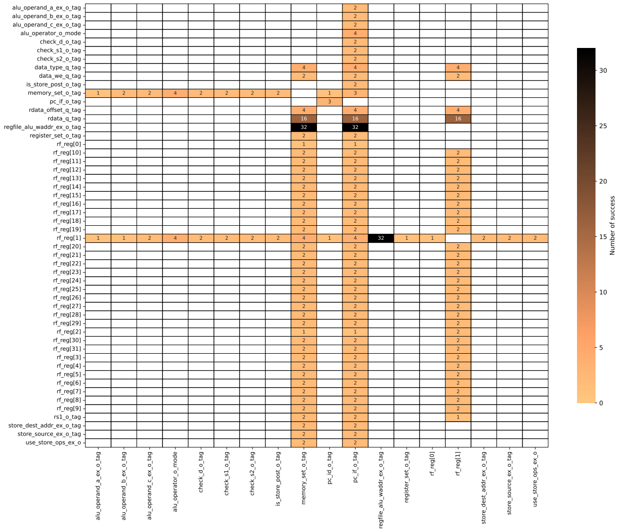


Figure 21: Unprotected version: multi-bits faults in two registers at a given clock cycle → 450 successes

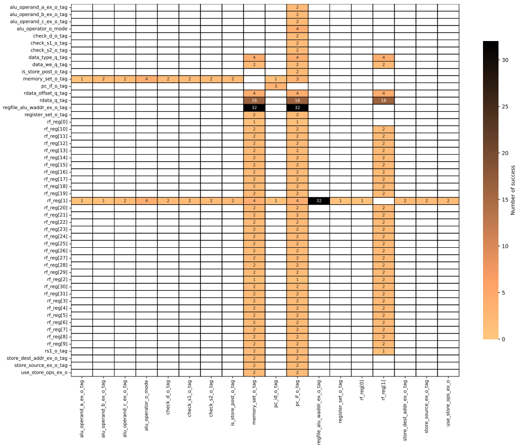


Figure 22: Unprotected version: multi-bits faults in two registers at a given clock cycle → 450 successes

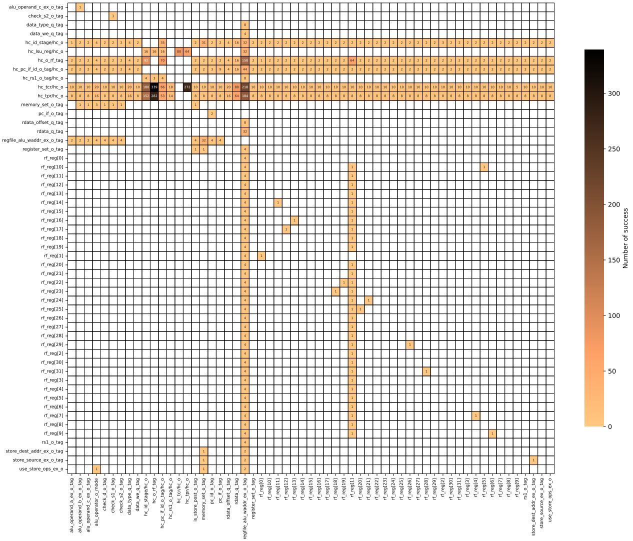


Figure 23: Hamming Code 2 protected version: multi-bits faults in two registers at a given clock cycle → 4356 successes

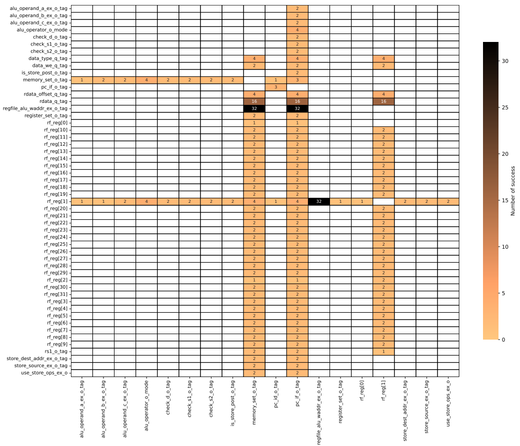


Figure 22: Unprotected version: multi-bits faults in two registers at a given clock cycle → 450 successes

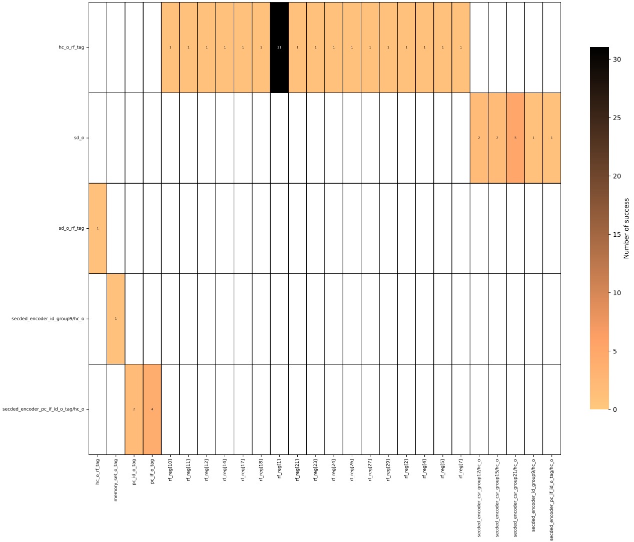


Figure 23: SECDED 5 protected version: multi-bits faults in two registers at a given clock cycle → 66 successes

V. Conclusion and Perspectives

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

Presented:

- Vulnerability assessment of a DIFT mechanism against FIA
  - We have shown that the DIFT mechanism is vulnerable
  - Presented different fault models adapted from the state-of-the-art to defeat the DIFT and its protections
- Open-Source tool to help find vulnerabilities during the conceptual phase. It enables the concept of *Security by Design*
- Proposition of 3 lightweight countermeasures:

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

Presented:

- Vulnerability assessment of a DIFT mechanism against FIA
- Open-Source tool to help find vulnerabilities during the conceptual phase. It enables the concept of *Security by Design*
- Proposition of 3 lightweight countermeasures:

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

Presented:

- Vulnerability assessment of a DIFT mechanism against FIA
- Open-Source tool to help find vulnerabilities during the conceptual phase. It enables the concept of *Security by Design*
- Proposition of 3 lightweight countermeasures:
  - based on parity codes
  - area overhead smaller than 8%
  - no impact on performances
  - good efficiency in terms of security

Short terms

- Propose more robust countermeasures to correct multiple faults
  - Evaluation of Reed-Solomon, BCH codes, or triplication
  - Evaluation of these countermeasures in terms of area and performances overhead compared to our actual proposed solutions
- Further development of FISSA
  - Better integration in the design workflow
  - More fault models
  - More configurability, for example, automatisation for finding targets
  - Adding a graphical user interface to provide a better experience

Perspectives

Short terms

- Propose more robust countermeasures to correct multiple faults
  - Evaluation of Reed-Solomon, BCH codes, or triplication
  - Evaluation of these countermeasures in terms of area and performances overhead compared to our actual proposed solutions
- Further development of FISSA
  - Better integration in the design workflow
  - More fault models
  - More configurability, for example, automatisation for finding targets
  - Adding a graphical user interface to provide a better experience

Long terms

- Conduct real-world FIA
  - Evaluation against clock glitches (ChipWhisperer [21]), EMFI (ChipShouter [22]), laser (ALPhANOV laser [23]) for examples.
- Extend the assessment of more complex DIFT
  - Evaluation of DIFT with more bits in the tag (e.g: Raksha [11] : 4-bit tags)
  - Evaluation of our proposed protections for these DIFT and comparison with other protections

International peer-reviewed conferences with proceedings

1. **William Pensec**, Vianney Lapôtre, and Guy Gogniat. 2023. 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), 2023. **Best paper award** [16]
2. **William Pensec**, Francesco Regazzoni, Vianney Lapôtre, and Guy Gogniat. Defending the Citadel: Fault Injection Attacks Against Dynamic Information Flow Tracking and Related Countermeasures. 2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2024, pp. 180-185. [19]
3. **William Pensec**, Vianney Lapôtre, and Guy Gogniat. Scripting the Unpredictable: Automate Fault Injection in RTL Simulation for Vulnerability Assessment. 2024 27th Euromicro Conference on Digital System Design (DSD), 2024. [18]

Source code

- **William Pensec**, FISSA: Fault Injection Simulation for Security Assessment, `https://github.com/WilliamPsc/FISSA`

Popularising science event

- Participation in a science outreach event, “*Ma thèse en 180 secondes*” (“My thesis in 180 seconds”), Rennes, March 2023, `https://youtu.be/m_whL8xGbMQ`

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

William PENSEC

Université Bretagne Sud, UMR 6285, Lab-STICC, Lorient, France

Thank you for your attention.

Composition of the Jury

Reviewers:	Lejla BATINA
	Vincent BEROULLE
	Nele MENTENS
Examiners:	Jean-Max DUTERTRE
	Francesco REGAZZONI
PhD supervisor:	Guy GOGNIAT
PhD co-director:	Vianney LAPÔTRE

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] 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.

[13] 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.

[14] 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.

References

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

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

References

[23] 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.

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

Backup

- Static or **Dynamic**
- Software, **Hardware** or Hybrid

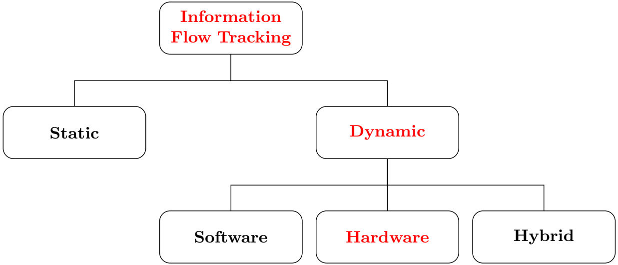


Figure 24: Taxonomy of IFTs

Tag Propagation Register

Table 8: 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 9: 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

Table 10: Logical fault injection simulation campaigns results for exhaustive multi-bits faults in one register 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	927	6	–	–	–	3 (0.32%)	936	00:08
	Simple parity	0	498	0	498	–	–	0	996	00:14
	Hamming 1	0	0	20	–	1962	–	10 (0.50%)	1992	00:28
	Hamming 2	0	0	12	–	2038	–	14 (0.68%)	2064	00:32
	Hamming 3	0	0	12	–	2352	–	12 (0.51%)	2376	00:28
	Hamming 4	0	0	12	–	2712	–	12 (0.44%)	2736	00:35
	Hamming 5	0	0	12	–	2976	–	12 (0.40%)	3000	00:45
	SECDED 1	0	0	8	–	1393	648	3 (0.15%)	2052	00:30
	SECDED 2	0	0	5	–	1475	666	2 (0.09%)	2148	00:30
	SECDED 3	0	0	4	–	1932	726	2 (0.08%)	2664	00:40
	SECDED 4	0	0	0	–	2370	822	0	3192	00:45
	SECDED 5	0	0	0	–	2670	798	0	3468	00:55

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));
}
```

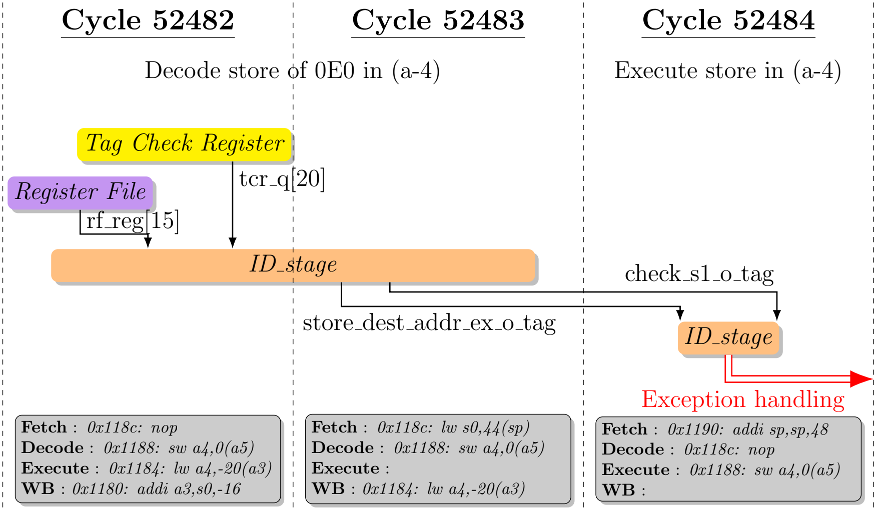


Figure 25: Temporal analysis of the tags propagation in a *format string* attack

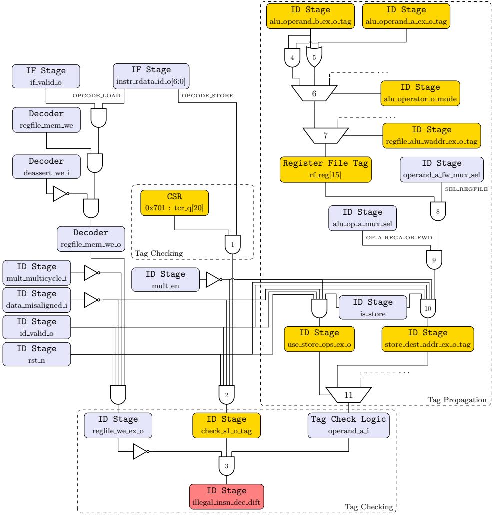


Figure 26: Logical analysis of the tags propagation in a *format string* attack

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)
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 12: Logical fault injection simulation campaigns results for exhaustive multi-bits faults in one register 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	1202	32	–	–	–	14 (1.12%)	1248	01:24
	Simple parity	0	661	0	665	–	–	2 (0.15%)	1328	02:12
	Hamming 1	0	0	62	–	2565	–	29 (1.09%)	2656	04:24
	Hamming 2	0	0	53	–	2666	–	33 (1.20%)	2752	03:36
	Hamming 3	0	0	47	–	3090	–	31 (0.98%)	3168	03:55
	Hamming 4	0	0	47	–	3570	–	31 (0.85%)	3648	04:25
	Hamming 5	0	0	41	–	3930	–	29 (0.73%)	4000	05:18
	SECDED 1	0	0	22	–	1832	864	18 (0.66%)	2736	03:30
	SECDED 2	0	0	14	–	1938	894	18 (0.63%)	2864	03:48
	SECDED 3	0	0	10	–	2560	968	14 (0.39%)	3552	04:42
	SECDED 4	0	0	5	–	3146	1096	9 (0.21%)	4256	05:42
	SECDED 5	0	0	4	–	3554	1064	2 (0.04%)	4624	06:30

Case 2: WU-FTPd

Table 13: 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

Case 3: Compare/Compute

- No software vulnerability
- Used to cover the DIFT surface

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

Case 3: Compare/Compute

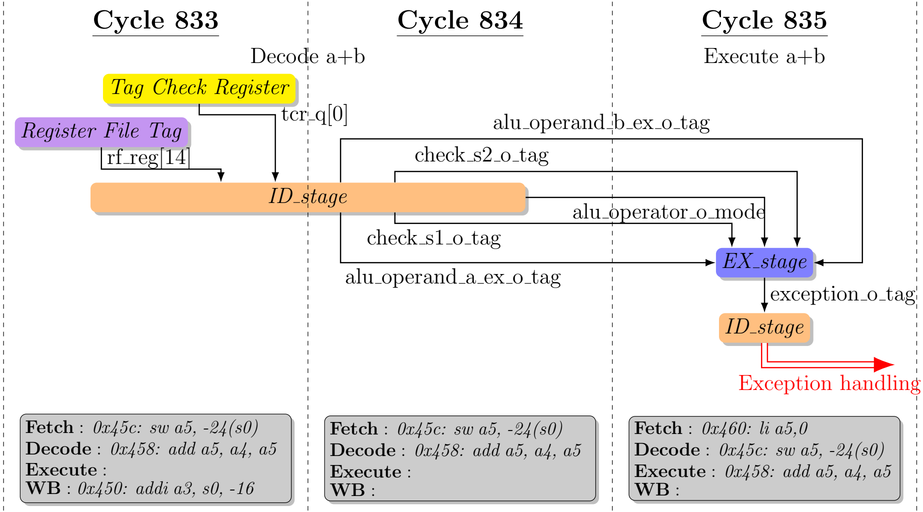


Figure 27: Temporal analysis of the tags propagation in a *format string* attack

Case 3: Compare/Compute

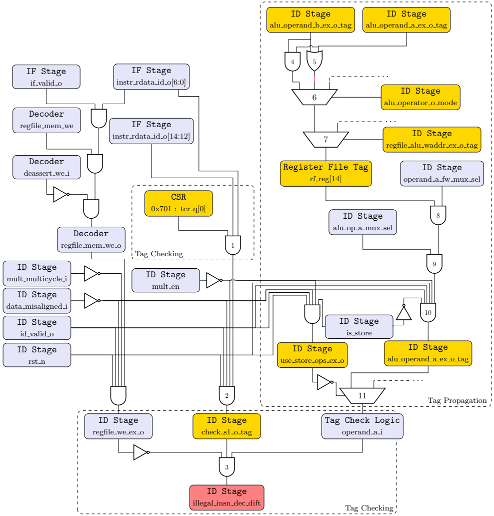


Figure 28: Logical analysis of the tags propagation in a *format string* attack

Case 3: Compare/Compute

Table 14: 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)
Compare Compute	No protection	0	29 906	919	–	–	–	1179 (3.68%)	32 004	05:24
	Simple parity	0	6697	202	27 678	–	–	7 (0.02%)	34 584	04:48
	Hamming 1	0	0	450	–	45 192	–	262 (0.57%)	45 904	09:21
	Hamming 2	0	0	440	–	48 419	–	125 (0.26%)	48 984	08:47
	Hamming 3	0	0	315	–	72 140	–	125 (0.17%)	72 580	13:53
	Hamming 4	0	0	97	–	103 345	–	70 (0.07%)	103 512	22:23
	Hamming 5	0	0	96	–	115 511	–	73 (0.06%)	115 680	23:48
	SECDED 1	0	0	0	–	37 740	11 244	0	48 984	17:00
	SECDED 2	0	0	0	–	46 236	7228	0	53 464	10:12
	SECDED 3	0	0	0	–	85 584	6436	0	92 020	18:25
	SECDED 4	0	0	0	–	136 040	4940	0	140 980	28:37
	SECDED 5	0	0	0	–	151 212	5028	0	156 240	32:52

Table 15: Logical fault injection simulation campaigns results for exhaustive multi-bits faults in one register at a given clock cycle

		Crash	Silent	Delay	Detection	Detection & Correction	Double Error Detection	Success	Total	Execution time (h:min)
Compare Compute	No protection	0	616	2	–	–	–	6 (0.96%)	624	00:04
	Simple parity	0	330	0	334	–	–	0	664	00:04
	Hamming 1	0	0	9	–	1311	–	8 (0.60%)	1328	00:09
	Hamming 2	0	0	15	–	1356	–	5 (0.36%)	1376	00:09
	Hamming 3	0	0	12	–	1567	–	5 (0.32%)	1584	00:11
	Hamming 4	0	0	12	–	1807	–	5 (0.27%)	1824	00:13
	Hamming 5	0	0	12	–	1983	–	5 (0.25%)	2000	00:14
	SECDED 1	0	0	2	–	888	476	2 (0.15%)	1368	00:09
	SECDED 2	0	0	6	–	977	449	0	1432	00:10
	SECDED 3	0	0	2	–	1290	484	0	1776	00:12
	SECDED 4	0	0	0	–	1580	548	0	2128	00:15
	SECDED 5	0	0	0	–	1780	532	0	2312	00:16

Table 16: 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)
Compare Compute	No protection	0	44 444	323	–	–	–	865 (1.90%)	45 632	05:36
	Simple parity	0	16 033	53	35 943	–	–	3 (0.01%)	52 032	08:05
	Hamming 1	0	0	2912	–	196 958	–	2690 (1.33%)	202 560	34:17
	Hamming 2	0	0	4677	–	211 969	–	1978 (0.90%)	218 624	37:24
	Hamming 3	0	0	4377	–	290 302	–	2089 (0.70%)	296 768	53:50
	Hamming 4	0	0	5282	–	393 423	–	2447 (0.61%)	401 152	74:31
	Hamming 5	0	0	5829	–	475 987	–	2664 (0.55%)	484 480	94:21
	SECDED 1	0	0	656	–	92 123	122 731	490 (0.23%)	216 000	35:42
	SECDED 2	0	0	1452	–	112 110	124 659	3 (0.0013%)	238 224	43:38
	SECDED 3	0	0	640	–	200 702	175 871	3 (0.0008%)	377 216	72:32
	SECDED 4	0	0	68	–	304 920	246 033	3 (0.00054%)	551 024	109:22
	SECDED 5	0	0	96	–	384 572	267 665	3 (0.00046%)	652 336	128:21