Processor Extensions for Hardware Instruction Replay against Fault Injection Attacks

Noura Ait Manssour, Vianney Lapotre, Gogniat Guy, Arnaud Tisserand

▶ To cite this version:

Noura Ait Manssour, Vianney Lapotre, Gogniat Guy, Arnaud Tisserand. Processor Extensions for Hardware Instruction Replay against Fault Injection Attacks. DDECS: 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, Apr 2022, Prague, Czech Republic. hal-03599317

HAL Id: hal-03599317
https://hal.science/hal-03599317v1

Submitted on 7 Mar 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Processor Extensions for Hardware Instruction Replay against Fault Injection Attacks

Noura AIT MANSSOUR[1,3], Vianney LAPÔTRE[2,3], Guy GOGNIAT[2,3] and Arnaud TISSERAND[1,3]
CNRS[1], Université Bretagne Sud[2], Lab-STICC[3] UMR 6285
Centre Recherche UBS, rue St Maudé, CS7030, 56321 Lorient, France

***Abstract*—The paper explores hardware supports for replaying instructions to protect processors against some fault injection attacks. A replay instruction is added to the instruction set of a small 32-bit RISC processor to allow the automatic and parametrized replay of sequences of instructions. Various detection elements are added to the processor, implemented on FPGA, and compared in terms of performances, cost and fault coverage. The proposed extension leads to significant improvements compared to software protections for a small silicon overhead.**

***Index Terms*—processor architecture; instruction set architecture extension; physical attack; hardware countermeasure.**

I. Introduction

Embedded processors can be subject to *physical attacks* due to some proximity between an attacker and the circuit. *Side channel attacks* [1] exploit correlations between observable parameters (e.g., computation time, power consumption, electromagnetic radiation) and secret data. *Fault injection attacks* (FIAs) [2], [3] exploit perturbations (e.g., glitch on power supply or clock, electromagnetic radiation, laser shot) in the circuit to reveal secret data [4] or bypass security features [5]. Below, we deal with hardware protections against FIA builtin in embedded processors.

Various protection methods exist against FIA (see [2] for an introduction): error code correction/detection, algebraic/functional property check, information and temporal redundancy, randomization, etc. *Redundancy* based solutions are popular in hardware and software.

In software (SW), *instruction duplication* and *triplication* [6], [7] are easy to use to secure critical (parts of) codes but lead to important overheads in execution time and code size. More, software protections rarely take into account hardware implementation details, such as the processor pipeline, and may not be as effective as intended. Sec. II quickly reviews some previous works on this topic.

Hardware (HW) support for *instruction replay* in processors is well known to resolve speculation issues in high-performance processors [8], improve fault-tolerance [9] and security [10]. In this paper, we propose *processor extensions* for *hardware instruction replay* in Sec. IV. One dedicated *replay instruction* is added to the instruction set of a small RISC processor described in Sec. III. Several *protection elements* are also added to the processor, see Sec. IV, and implemented on FPGA, see Sec. VI.

We evaluate various protected versions of our processor and compare them to typical software protections in terms of cost, speed and robustness against FIA using logical simulations on typical critical benchmarks in Sec. VI.

In this paper, we only protect the *data flow* but not the control flow (see [11], [12]). Our team works on this topic (see [13] for a recent PhD) and we plan to add protections of the control flow in the future.

II. State of the Art

Faults are identified as a *reliability* issue in processors since the 70s [14]. FIAs also lead to *cybersecurity* issues [2], [3]. Protections based on *redundant computation or information* are available in hardware and software. In hardware for instance, double data rate computation [15] allows high coverage but requires larger circuits with a lower frequency. In software, *temporal redundancy* is popular with methods from manual instruction duplication and triplication [6], [7] to compiler-assisted solutions [16].

Instruction duplication with comparison and instruction triplication with voting from [6] lead to important overheads in terms of execution time (resp. 2x to 4x) and code size (resp. 4x to 14x per protected instruction). These overheads are not the only concerns with such protections. [6] assumes that a single fault only impacts a single instruction during the execution, but this is not the case. [17] and [18] show that in pipelined processors, a single fault impacts several pipeline stages and then several instructions, not only one. These software protections can be bypassed even with a single fault [19], [20], [21].i

III. Target Processor

Our processor is a small homemade 32-bit RISC designed, in System Verilog, for exploring hardware protections at architecture level. It is close to a RV32IM RISC-V core [22] with a slightly different branching system and less features. We plan to port our protections to RISC-V. Fig. 1 presents a simplified schematic of the processor architecture. The pipeline has 2 stages: fetch | decode + execute. The register file (RF) has 32 registers of 32 bits each with 2 read ports and 1 write port. Register `R0` is fixed at the value 0. The processor contains 2 execution units for 32-bit integers: one arithmetic and logic unit (ALU) and one multiplier. The load/store unit (LSU) accesses the data-memory hierarchy (`D-MEM`) without cache for this paper. The instruction memory `I-MEM` is also without cache. The processor also includes hardware performance counters for cycles, instructions, memory accesses, branches,

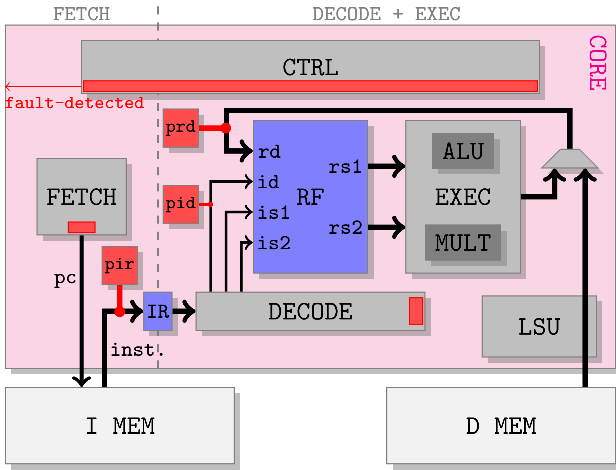

Fig. 1. Simplified schematic of the processor architecture (not all signals and elements are represented). Red elements are for replay protection.

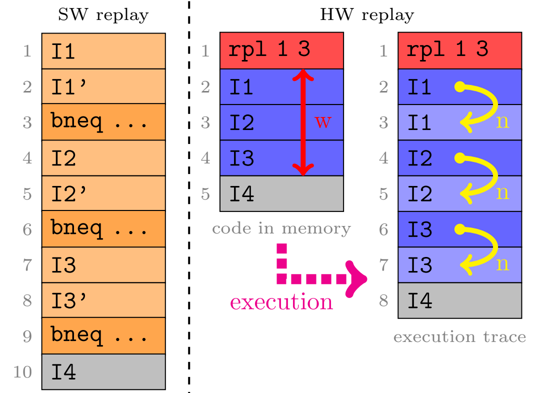

Fig. 2. Illustration of a typical software protection and our hardware replay protection in a small 4-instruction code ('I1-4' where 'I1-3' are protected using one replay and 'I4' is not).

and protections monitoring. It is fully implemented and validated on FPGA (see Sec. V).

We also develop an assembler to generate binaries and scripts for cycle accurate and bit accurate HDL simulation for functional validation and fault injection evaluation.

IV. Proposed Hardware Replay Protection

Our protection consists in a *replay instruction* added to the instruction set, various *detection elements* added to the core and *modifications* in the processor control detailed in this section. See Sec. VI for implementation results and performance evaluation.

A. Replay Instruction

The replay instruction, named `rpl`, is intended to protect small critical sequences of instructions, called *replay windows*. It takes 2 arguments, `rpl n w`, and specifies that each instruction in the window of the `w` instructions immediately following `rpl` will be *executed once and replayed* `n` times for a total of `n`+1 consecutive executions.

This behavior is illustrated in the right side of Fig. 2 with a simple 4-instruction example: `I1`, `I2`, `I3`, `I4`. Instructions `I1`, `I2`, `I3` must be protected and executed twice while `I4` is kept unprotected. The window of instructions to be protected has a width `w=3` and the number of replay(s) is `n=1`. Thus, a replay instruction `rpl 1 3` is added just before the instructions window `I1-3`.

Left side of Fig. 2 presents the same example in case of a pure software replay protection for comparison.

B. Detection Elements

Just replaying instructions may not be sufficient. The result of each replayed instruction has to be *compared* to the original one. For this, we add *detection elements* (DE) depicted in Fig. 3. At each replay cycle, a DE compares the current `value` with the original one stored at the first execution of the instruction. The `enable` signal on the register is managed by the processor control. The comparison result `diff` feeds the processor control to signal different values during replay cycles. In such a case, a specific `fault-detected` interruption signal is raised. The security policy related to this interruption is out of the scope of this paper.

To protect the data flow in the processor, we protect the result from the execution units and LSU. The 32-bit DE called `prd` in Fig. 1 protects the destination register (`rd`) written in RF. The index `id` of the destination register is protected by the 5-bit DE called `pid`. The `pir` DE protects the instruction register (see Sec. IV-D for details).

Compared to software replay, a cheap `prd` DE (one register and comparator) avoids the use of additional registers in the register file and explicit detection instructions in the code. In software replay Fig. 2 (left side), instruction `I1` (line 1) is replayed by instruction `I1'` (line 2) with a different destination register (but similar opcode and sources), and the comparison in line 3 compares the two result registers and branches to the software error handler if they are different. Duplication with comparison in software leads to 3 executed instructions for each protected instruction. This can be 4 in processors where comparison and conditional branch are separated instructions. Our hardware replay leads to smaller codes in memory and fewer executed instructions. The number of original and replayed instructions is the same as for software replay. But there is no comparison instruction(s) in the code but one `rpl` instruction for each replay window. The window width `w` in the `rpl n w` instruction leads to fewer instructions as soon as `w`>1 compared to software replay.

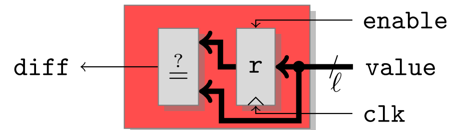

Fig. 3. Detection element (used for `prd`, `pid`, `pir` in Fig. 1).

C. Processor Modifications for Replay Support

The processor control is modified to handle our hardware replay. A few internal flip-flops and logic gates are added to control the replay in the pipeline. When a `rpl` is decoded, the processor stores `n` and `w` into small local registers. During the execution of an instruction in a replay window, small counters control the number of replays and the position in the window. For each DE, the control must enable its register during the first execution of an instruction. During replay cycles, the control checks for `diff` signals to raise the `fault-detected` interruption.

The behavior of the processor is not modified at the ISA level. The addition of the `rpl` instruction does not change the behavior of the other instructions. This allows users to protect their codes with small additions on the assembly code. The automation of this protection for higher level codes (e.g., C) is out of the scope of this paper.

D. Processor Versions

Several *versions* of the processor with different protections solutions are evaluated in Sec. VI. The maximal width of a replay window `w` is limited by a hardware parameter, called Ω, in our HDL code such that $1 \leq \mathtt{w} \leq 2^{\Omega}$. We study the impact of the replay window width on cost and performance for $\Omega \in \{1, 2, 3, 4\}$ (see Sec. VI for Ω determination).

Possible values for `n` are in $\{1, 2, 3, 4\}$ leading to 2-bit registers and counters in the control for the number of replays. We use 2-bit values for `n` since we target single fault protection. Protection against injection with multiple faults close in time and space will be a challenge for future systems to avoid a prohibitive cost due to numerous replays.

The `prd` and `pid` DEs are intended to protect the destination register from execution units and LSU. To complement this type of data flow protection, we also study replay support with a DE called `pir` in Fig. 1 to protect the *instruction register* (`IR`) fed from the instruction memory. It protects the indexes for source registers `is1`, `is2` as well as the index of the destination register. Protecting the 3 indexes `id`, `is1`, `is2` using individual DEs would cost $3 \times 5 = 15$ bits (about half of the 32 bits in `pir`). Adding a specific DE for immediate operands in the instruction would also add up to the protection bill. Then adding `pir` may be a good way to protect all parts of the instruction directly for a small overhead.

We also design a version with a *refetch* support. During the replay cycles, the current protected instruction is refetched from the instruction memory and compared to the original one. This is simple in a 2-stage pipeline, but this would be more costly for deeper ones. `pir` and refetch are different. Refetch leads to detect more faults in the interface between the core and `I-MEM` but costs more energy. We plan to study energy aspects of our protections in a future work.

V. Evaluation Environment

HDL codes are implemented in a Zynq 7020 FPGA using Vivado V2018.3 from Xilinx after functional validation in simulation. In the next sections, we report post-synthesis and place&route results for area (LUTs, flip-flops) and performance at the fastest obtained clock frequency for each version of the processor. We evaluate various versions of the processor and protection elements proposed in this paper:

- `wop`: base processor *without any protection* (implementation results at line V=1 in Tab. I);
- `wphw_x`: base processor *with protection* for different DE selections and parameters where $\mathtt{x} \in \{2, 3, \cdots, 11\}$ (see lines V=`x` in Tab. I for details).

All the processor versions require the exact same number of BRAMs (16) and DSP blocks (4). Since they are only used in `I-MEM`, `D-MEM` and execution units, they are not impacted by the replay support and DEs.

To compare to pure software replay, we also evaluate a version called `wpsw` with the base processor `wop` and using *software protection* from state of the art [6], [7]. This is not a hardware version, the processor is the one in `wop`, only the code is changed.

To evaluate and compare these versions over various representative binary codes, we protect and execute the following functions commonly used in parts of secure applications:

- `verify_pin` to authenticate users (tested with 4 digits and a constant time code);
- `memcpy` to copy a memory region (critical for sensitive data, tested for various lengths);
- `atoi` to convert a string to an integer (critical when reading some identifiers, tested for various lengths);
- `trivium` a stream cipher used in some cryptographic applications (to reduce simulation time, we only focus on its `generate_bit` function which computes a new result bit and update the internal state).

We plan to use benchmarks such as FISSC [23] in the future.

For evaluating our protections, we use logical fault injection simulations on the HDL description (at cycle accurate and bit accurate level) of the processor. We inject faults for numerous *injection times*, *locations* and *types* as illustrated in the pseudo algorithm below:

```
list_codes = [verify_pin, memcpy, atoi, trivium]
list_versions = [wop, wpsw, wphw_1, wphw_2, ...]
list_fault_types = [stuck0, stuck1, bit_flip]

for code in list_codes:
  for version in list_versions:
    trace = get_trace(code, version)
    for inst in get_inst(trace):
      for reg in get_reg(inst, version):
        for type in list_fault_types:
           ftrace = insert_fault(trace, inst, reg, type)
           state = simulate(version, ftrace)
           save(state)
```

For each `code` and `version`, the `get_trace` function provides the execution `trace` corresponding to `code` adapted for the processor `version`. For a version with software protection, explicit replayed instructions (e.g. `I1'` for `I1` in Fig. 2 left side) and comparison instructions (e.g. `bneq`) are added. For a version with hardware protection, `rpl`

instructions are added. When the `version` is unprotected, there is no addition to base `code`.

We use a large selection of executed instructions related to the data flow as *injection times*: all instructions before and after a loop in the function, and all instructions in the first, intermediate and last iterations of a loop (to save simulation time). The `get_inst` function provides the list of all instructions where a fault will be injected.

As *injection locations* we use all registers of the base processors and all registers of our protection elements (DEs and replay control). This is the purpose of the `get_reg` function. We only inject faults on registers used by the `inst` instruction (IR, RD in RF, registers in our hardware protections). In this work, we only inject faults on registers since several physical faults lead to setup/hold time violation in flip-flops [19], [20], [21]. We plan to extend this with other types of faults in the future (logical simulation may not be sufficient).

To limit simulation time, we only use 3 types of *injection types*: stuck at 0 or 1, or one bit flip at a random position of the faulted register.

When a `code`, `version`, instruction `inst`, faulted register `reg` and `type` are selected, the `insert_fault` function generates the HDL script for the simulation of this fault in the complete trace. There is one simulation for each fault time (`inst` instruction in `trace`), location (`reg` register) and `type`. Then, the simulation is performed and its state (memory, all registers contents and flags) is saved for analysis.

For each version of the processor and code, a reference trace is also required for the execution of the base code without any fault injection. This reference trace is used to compare the simulation `state` of each simulation with a fault injection. This reference simulation is not represented in the pseudo algorithm to simplify the presentation.

The efficiency of a protection solution is evaluated using its *detection rate*. For the software protection, there is a detection when the detection code branches to the error handler instead of continuing the nominal execution. For our hardware protection, the detection signals are provided by the DEs.

The various loops iterations in the pseudo-algorithm above correspond to dozens of thousands of simulations over a few weeks of CPU time. Actual numbers of simulations are reported in gray in Tab. II.

As this paper targets the protection of the data flow, we do not attack the control flow of the pure software protection and applications. But we inject faults in all registers of our protections: both "data" registers in DEs and control registers for replay management.

VI. Evaluation of our Hardware Protections

Below, we report and analyze results for FPGA implementation, execution time, code size and fault injection simulations. We close the section with additional discussions.

A. FPGA Implementations Results

Tab. I reports FPGA results for all versions of the processor and various protection elements configurations (see Sec. IV-D and Sec. V for details). Each line in the table is a version denoted Vα with $\alpha \in [1, 11]$. "Protection" columns indicate the configuration of the protection elements in the version. Gray values are relative differences $(x - ref)/ref$ w.r.t. reference values in V1 expressed in %. Version 1 is the *reference processor* without hardware replay or detection support.

TABLE I
FPGA (Zynq 7020) implementation results for various versions (V) of the processor and protection elements.

V	Protection						Area				Freq.	
	HW	Ω	pid	prd	ref	pir	LUT	%	FF	%	MHz	%
1	✘	✘	✘	✘	✘	✘	731	+00	301	+00	304	+00
2	✓	4	✘	✘	✘	✘	756	+03	324	+08	294	-03
3	✓	4	✓	✘	✘	✘	759	+04	329	+09	301	-01
4	✓	4	✘	✓	✘	✘	799	+09	356	+18	297	-02
5	✓	4	✓	✓	✘	✘	801	+10	361	+20	294	-03
6	✓	3	✓	✓	✘	✘	799	+09	359	+19	300	-01
7	✓	2	✓	✓	✘	✘	797	+09	357	+19	303	-00
8	✓	1	✓	✓	✘	✘	797	+09	355	+18	305	+00
9	✓	4	✘	✘	✓	✘	757	+04	325	+08	295	-03
10	✓	4	✘	✓	✓	✘	797	+09	357	+19	287	-06
11	✓	4	✘	✓	✓	✓	809	+11	389	+29	283	-07

Version 2 adds replay support but no detection for a very small impact: +10% area and -3% clock frequency.

Versions 3 to 5 add various combinations of detection elements to V2. Between V2 and V3, the 5 additional flip-flops are for the 5 bits in `pid`. Between V2 and V4, the 32 additional flip-flops are for the 32 bits in `prd`. V5 costs 37 (32 + 5) flip-flops more than V2 due to `pid` and `pir`. The area penalty is +20% for V5 but those DEs do not change much the clock frequency.

Versions 5 to 8 only differ in the maximal window width supported ($\mathtt{w} \leq 2^{\Omega}$). Its impact is negligible for $1 \leq \Omega \leq 4$.

Versions 9 to 11 add refetch support (column "ref"). V10 and V11 add `prd` and `pir` support to V9 for a more complete protection. There is no `pid` in these versions since `id` is already in the refetched instruction (and in `pir` in V11). The impact is still limited: +30% area and -7% clock frequency.

In the following, we only use V11, the most advanced version of the processor, for the other evaluations (performance, code size, fault injection simulations) due to space constraints and limited benefit of the smaller versions.

Adding hardware support for replay and detection support does not cost much. The +30% area overhead in a very simple core should be even smaller in more complex ones. In cores with a deeper pipeline, hardware replay will require more internal registers and slightly more complex control than in our 2-stage one. But we think that the overall penalty should be still limited. We still have to evaluate if some internal resources can be shared between deeper pipeline elements and hardware replay support (and DEs).

B. Timing Performance and Code Size Results

Fig. 4 presents the performance results in terms of execution time and code size for each evaluated code (`verifyPin`, `memcpy`, `atoi` and `trivium`) used in 3 cases:

- reference: unprotected code on base processor V1;
- SW protection: protected code with software replay from state of the art [6], [7] on base processor V1;
- HW protection: protected code with our hardware replay on processor V11.

For both SW and HW protections, protected instructions are replayed once (`n=1` for our `rpl` instructions).

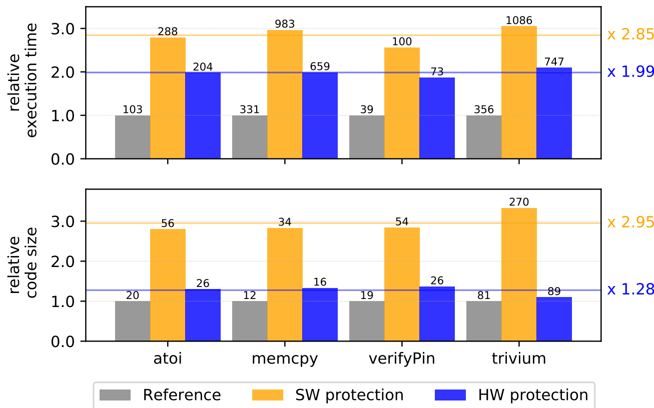

Fig. 4. Execution time and code size results. Values above the columns are: the number of cycles (top); and the number of instructions (bottom). Values on the right are averages.

The benefit of a hardware replay is clear compared to a software one as illustrated in Fig. 2. The execution time is reduced from a factor almost 3 with SW replay to 2 with HW replay. Going well below 2 for `n=1` (more generally below `n+1`) will be difficult since the processor must execute `n+1` times each protected instruction (unless using parallelism [10], [14]). The code size is also significantly reduced from a factor 3 with SW replay to only 1.3 with HW replay. This may constitute a potential interest in embedded systems.

C. Fault Injection Simulations Results

Using real fault injections would be interesting, but this is not simple in a FPGA implementation of softcore processors since one does not really know how close are the fault effects in the FPGA compared to a hardcore processor in an ASIC. We use logical fault injection simulations at HDL level as explained in Sec. V. Tab. II reports the corresponding results for 2 cases:

- SW protection: protected code with software replay from state of the art [6], [7] on base processor V1;
- HW protection: protected code with our hardware replay on processor V11.

codes	protection solutions			
	software		hardware	
`verifyPin`	0.757	(969)	0.760	(3036)
`memcpy`	0.743	(1341)	0.796	(3954)
`atoi`	0.764	(1317)	0.790	(6897)
`trivium`	0.737	(3222)	0.772	(9195)
mean	0.750		0.779	
std. dev.	0.011		0.014	

TABLE II

FAULT INJECTION SIMULATION RESULTS. THE DECIMAL VALUES ARE RATIOS OF NUMBER OF EXECUTIONS WHERE THE PROTECTION SOLUTION (SW OR HW) DETECTS A FAULT OVER THE TOTAL NUMBER OF EXECUTIONS (INTEGERS IN GRAY) FOR EACH CODE.

The hardware replay leads to similar fault detection results than the software one (about 75% of faults detected). But this protection level is achieved with much faster execution times and smaller codes for a moderate hardware overhead. This shows that hardware replay can be an interesting solution for protecting the data flow.

One should keep in mind that this fault injection simulation campaign is in favor of the software protection since we do not inject faults in the control flow (e.g. all `bneq` instructions in Fig. 2) while we inject faults in all the control and detection registers of our hardware replay. We will complete those simulations when working on the control flow protection.

D. Additional Discussions

To help to determine a good Ω value for the maximal width of replay windows ($\mathtt{w} \leq 2^{\Omega}$), Tab. III reports the distribution of `w` values in protected source codes and their corresponding execution traces. The count of the number of times an actual window of width `w` occurs is reported as $\mathtt{w}^{(\text{count})}$. Depending on the actual input data of the code, replay windows in the source code can be executed different numbers of times or not executed at all in the execution trace due to conditional jumps and loops. The "in source" column represents the effort provided by the user to protect the code. The "in trace" column shows what are the values `w` used during actual executions (for given input data). Reported values in Tab. III ("in trace" column) are typical results. Most of time, only small `w` values are used. But in `trivium` example, there is a long 54-instruction sequence to be protected before the loop (4 `rpl` are required with $\Omega = 4$, since $54 = 3 \times 2^4 + 6$) and a 17-instruction one after the loop. As those sequences are only used outside of the internal loop, the impact of a Ω value smaller than $\lceil \log_2 54 \rceil$ is very small. Increasing Ω would not

codes	replay windows distributions $\mathtt{w}^{\text{count}}$	
	in source	in trace
`atoi`	$1^{(1)}, 2^{(1)}, 3^{(3)}, 4^{(1)}$	$1^{(1)}, 3^{(10)}, 4^{(8)}$
`memcpy`	$2^{(1)}, 3^{(1)}, 4^{(1)}$	$2^{(1)}, 3^{(1)}, 4^{(8)}$
`verifPin`	$1^{(4)}, 3^{(1)}, 4^{(2)}$	$1^{(4)}, 3^{(1)}, 4^{(5)}$
`trivium`	$1^{(2)}, 6^{(2)}, 16^{(4)}$	$1^{(35)}, 6^{(36)}, 16^{(4)}$

TABLE III

DISTRIBUTION OF THE WIDTHS OF REPLAY WINDOWS IN SOURCE CODES AND THEIR CORRESPONDING EXECUTION TRACES.

cost much (see lines V5-8 in Tab. I), but this may not be very useful. For instance, using $\Omega = 6$ would reduce the total number of `rpl` executed for `trivium` to (35-1)+(36-1)+2=71 instead of 75 for $\Omega = 4$. In the future, we will evaluate how this parameter impacts protections of the control flow. For the `atoi` code in Tab. III, one can notice that the replay window `w=2` in the source code "disappears" in the trace since it is not executed for the actual data set. We plan to evaluate more codes and data sets in the future.

Currently, we consecutively execute each instruction and its replay(s) before going to the next instruction in the window. To reduce the switching activity in the processor, we do not consider another solution: executing each instruction in the window, then replay them as a block of instructions.

Regarding energy aspect, we also have to investigate links between refetch and memory hierarchy.

VII. Conclusion and Future Prospects

We propose a *hardware support for instruction replay* in a small RISC processor. It consists in a small extension of the instruction set (one new instruction), a few detection elements (mainly registers and comparators) and light modifications of the processor control. We explore various configurations of internal protection elements for hardware replay.

The core area overhead is about +30% while the clock frequency is reduced by less than 10% on FPGA.

The hardware replay allows to significantly reduce the execution time and code size compared to a pure software replay protection (respective reductions: $\times 3 \rightarrow \times 2$ and $\times 2 \rightarrow \times 1.3$).

The protection efficiency is evaluated using numerous logical fault injections in HDL simulation. It is similar to the one of software replay solutions but with much smaller execution times and code sizes.

In the future, we plan to study deeper pipelines, protection of the `rpl` instruction itself, porting to a RISC-V processor and energy optimizations. We also plan to study hardware protections of the control flow and their combination with hardware replay. Links to secure software development, compilation and security policy management should also be studied. Finally, we would like to perform more complete fault injection simulations and real attacks for security evaluation.

Acknowledgments

This work has been partially supported by a PhD grant from Région Bretagne/Pôle de Recherche Cyber.

References

[1] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks. Revealing the Secrets of Smart Cards*. Springer, 2007.

[2] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, Feb. 2006.

[3] B. Yuce, P. Schaumont, and M. Witteman, "Fault attacks on secure embedded software: Threats, design, and evaluation," *Journal of Hardware and Systems Security*, vol. 2, no. 2, pp. 111–130, Jun. 2018.

[4] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, ser. LNCS, vol. 1233. Springer, 1997, pp. 37–51.

[5] A. Vasselle, H. Thiebeauld, Q. Maouhoub, A. Morisset, and S. Ermeneux, "Laser-induced fault injection on smartphone bypassing the secure boot," in *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, Aug. 2017, pp. 41–48.

[6] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented AES: Effectiveness and cost," in *Proc. Workshop on Embedded Systems Security (WESS)*. ACM, Oct. 2010, pp. 7:1–10.

[7] V. B. Thati, J. Vankeirsbilck, D. Pissoort, and J. Boydens, "Selective duplication and selective comparison for data flow error detection," in *Proc. International Conference on System Reliability and Safety (ICSRS)*. IEEE, Nov. 2019, pp. 10–15.

[8] I. Kim and M. H. Lipasti, "Understanding scheduling replay schemes," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Feb. 2004, pp. 198–209.

[9] K. A. Bowman, J. W. Tschanz, N. S. Kim, J. C. Lee, C. B. Wilkerson, S.-L. L. Lu, T. Karnik, and V. K. De, "Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance," *IEEE Journal of Solid State Circuits*, vol. 44, no. 1, pp. 49–63, Jan. 2009.

[10] R. Psiakis, "Performance optimization mechanisms for fault-resilient VLIW processors," PhD Thesis, Université Rennes 1, Dec. 2018. [Online]. Available: https://tel.archives-ouvertes.fr/tel-02137404

[11] M. Werner, E. Wenger, and S. Mangard, "Protecting the control flow of embedded processors against fault attacks," in *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*. Springer, Nov. 2015, pp. 161–176.

[12] R. de Clercq, "Hardware-supported software and control flow integrity," PhD Thesis, KU Leuven, Nov. 2017. [Online]. Available: https://www.esat.kuleuven.be/cosic/publications/thesis-297.pdf

[13] M. A. Wahab, "Hardware support for the security analysis of embedded softwares: applications on information flow control and malware analysis," PhD Thesis, Centrale Supelec, Université Bretagne Loire, Dec. 2018. [Online]. Available: https://tel.archives-ouvertes.fr/tel-02634340

[14] T. Li, J. A. Ambrose, R. Ragel, and S. Parameswaran, "Processor design for soft errors: Challenges and state of the art," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, pp. 57:1–44, Sep. 2016.

[15] P. Maistri and R. Leveugle, "Double-data-rate computation as a countermeasure against fault analysis," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1528–1539, Nov. 2008.

[16] J. Proy, K. Heydemann, A. Berzati, and A. Cohen, "Compiler-assisted loop hardening against fault attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, Dec. 2017.

[17] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software fault resistance is futile: Effective single-glitch attacks," in *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, Aug. 2016, pp. 47–58.

[18] B. Yuce, N. F. Ghalaty, C. Deshpande, H. Santapuri, C. Patrick, L. Nazhandali, and P. Schaumont, "Analyzing the fault injection sensitivity of secure embedded software," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 4, pp. 95:1–25, Jul. 2017.

[19] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, and E. Encrenaz, "Experimental evaluation of two software countermeasures against fault attacks," in *Proc. International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, May 2014, pp. 112–117.

[20] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, "On the importance of analysing microarchitecture for accurate software fault models," in *Proc. Euromicro Conference on Digital System Design (DSD)*. IEEE, Aug. 2018, pp. 561–564.

[21] ——, "Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor," *Microprocessors and Microsystems (MICPRO)*, vol. 71, pp. 1–10, Nov. 2019.

[22] D. Patterson and J. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017. [Online]. Available: https://www.elsevier.com/books/computer-organization-and-design-risc-v-edition/patterson/978-0-12-812275-4

[23] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "FISSC: A fault injection and simulation secure collection," in *Proc. International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*. Springer, Sep. 2016, pp. 3–11.