A hardware design methodology to prevent microarchitectural transition leakages

Mathieu Escouteloup[1][0009−0001−8254−9130] and Vincent Migliore[2][0009−0006−2194−7991]

[1] Université de Bordeaux, Bordeaux INP, Laboratoire IMS, UMR CNRS 5218, France first.last@ims-bordeaux.fr
[2] LAAS–CNRS, Univ. Toulouse, CNRS, INSA, Toulouse, France first.last@laas.fr

Abstract. Side-channel attacks allow information extraction from a system by analyzing indirect observations. For instance, power consumption is known to be correlated with sensitive data manipulated by digital components. Recent efforts have been put on securing the system at the software level with a formally proven method called masking. They rely on an abstract model of the target where automatic countermeasures can be efficiently applied. Recent work focused on microarchitecture, i.e. implementation details of the hardware, to deal with residual vulnerabilities which require strong knowledge of the system's hardware and have limited portability.
In this paper, we present a generic methodology to harden the processor's microarchitecture to allow straightforward software defense strategy implementation (like masking) with minimal knowledge of the hardware. Based on a fine-grained vulnerability diagnosis at the microarchitecture level and a generic design hardening strategy, our proposition can be applied to produce several processors with security, performance and area tradeoffs. In addition, we provide two secured designs based on a customizable RISC-V processor and its memories, validated with real measurements on an FPGA.

Keywords: Power side-channels · Design method · Processor design · Microarchitecture · RISC-V

1 Introduction

Cyberphysical and connected objects have gained momentum recently due to their increasing adoption to address societal challenges (smart cities, connected cars, smart factories...). To support flexibility and updatability requirements, modern connected objects are generally processor-based. They have also gained in complexity, with the integration of more and more components into a single chip, such as System-on-Chip (SoC), making security assessment a serious challenge. Compared to usual computer systems, they also face a large range

of physical attacks, which exploit some characteristics of the target system including both software and hardware. Among these, Side-Channel Attack (SCAs) can infer secret information from physical observations measured during the execution of sensitive computations. For instance, the AES encryption algorithm, considered as strongly secured, is highly vulnerable to SCAs.

For power observation, one of the well-known mitigation techniques is called masking [19]. The idea is to break down secret data into multiple values called *shares* that are individually statistically independent from the secret. For instance, to mask a secret s with boolean masking, a uniform value r is generated and XOR-ed with s, producing shares $(s \oplus r,\ r)$.

While theoretically unbreakable, masking suffers from weaknesses when implemented on a given target, especially when the system's model does not perfectly match the real hardware. For instance, in processor-based architecture, shares can be processed successively or simultaneously, leading to potential transitions between them, which can be directly exploited to reduce the global security level [5].

In practice, it has been proven that many registers or wires can generate transition between shares [5,23,12] meaning that processor's microarchitecture is a non-negligible source of leakage. Unfortunately, since they are deeply hidden in the microarchitecture, hardening software or hardware against power-based side-channel attacks is challenging. Several defense mechanisms have been proposed to increase the security level of masked implementations. Some of them try to patch the software directly by inserting basic instructions to clean internal hardware states, manually [26] or automatically [29,28]. The Instruction Set Architecture (ISA) can also be directly modified [16] to integrate some instructions for hardware cleaning. However, in addition to impacting performance, they also need a precise understanding of the microarchitecture issue to know which transitions must be mitigated. This information is not always accessible in the gray-box system or in available models [17]. Finally, another approach is to directly harden the processor. Data obfuscation and hardware masking [4,13,33] are then used to build a new security layer at the hardware level. Hardening is also possible physically directly using Electronic Design Automation (EDA) tools [31]. However, all these strategies finally try to hide the transition leakages.

Contributions Based on all these observations, we propose to address the transition leakage issue from its root cause: the microarchitecture. Instead of trying to modify existing software and hardware by target-specific countermeasures, we propose a generic design approach to mitigate transition leakages from microarchitecture's description. Using the properties of a high-level hardware description language (here Chisel), we develop a set of fundamental blocks where we apply most of our generic design strategies. They are then used to build a complete microarchitecture with security properties, limiting processor-specific changes. Finally, the whole strategy is analyzed using RTL simulations and then validated by performing real measurements on a Field-Programmable Gate Array (FPGA) target.

More generally, this works aims at addressing microarchitectural issues at that abstraction layer. It is part of a global security strategy where each layer of the system must be considered. In that way, other hardening strategies (masking for the software abstraction layer, or glitches for the physical layer) can be combined to achieve a given security level.

This approach offers new hardware designer directives to secure its design, allowing the removal of potential weaknesses before the fabrication. This methodology can be summarized in four steps, iterated to compare multiple hardened designs:

1. generate the associated Register Transfer Level (RTL) description from the target microarchitecture specification,
2. build a leakage model considering all data registers and signals are potential leakage sources,
3. evaluate in both the simulation and real target the security using a set of microbenchmarks to reproduce all the data transitions,
4. enrich the microarchitecture description using generic hardening strategies to remove the leaky transitions.

The rest of this paper is organized as follows. Section 2 provides a description of the threat model, the leakage model and a description of known hardware vulnerabilities. Section 3 presents our generic methodology in the design phase. Section 4, Section 5 and Section 6 provide a complete analysis and instantiation of the methodology to a processor-based microarchitecture, with a focus on microarchitecture description and RTL generation in Section 4, the required microbenchmarks for the security assessment in Section 5, and how to apply the generic countermeasures in Section 6. In Section 7, a validation of the hardened processor is proposed using both simulation and real measurements on an FPGA target.

2 Background

2.1 Threat Model

In this paper, we consider the side-channel scenario where the execution and manipulation of sensitive data impact its physical environment. Particularly, we are interested here in power consumption variations. Then, in our scenario, we consider that an attacker has physical access to its target (*e.g.* an Internet-of-Things (IoT) device with a processor). He also has the different necessary equipment to perform power consumption measurements. An attack is considered successful if a malicious person is able to observe variations which directly depend on sensitive data in the target. In the other way, an efficient defense strategy must ensure that an attacker will not be able to recover any information from sensitive data by analyzing power consumption traces. We will see in Subsection 2.3 that it concerns the great majority of the potential leakages.

2.2 Transition Leakages

A transition is a value change in a wire or register from a data $d1$ to $d2$. Basically, at the circuit level, a bit flip ($0 \rightarrow 1$ or $1 \rightarrow 0$) in the system leads to a temporary consumption increase. Then, when a transition between two sensitive values occurs, it impacts the consumption proportionally to the Hamming Distance between $d1$ and $d2$. In the case of masking [20,26,15,22,19], consider a secret value s, a mask m and the masked value v, then: $v = m \oplus s$. Basically, only m and v are directly manipulated. This is what is done by some secure cryptographic implementations [10]. However, if a transition $m \rightarrow v$ occurs, the Hamming Weight will probably leak while it is directly correlated to s. By performing multiple measurements, it leads to a potential reduction of the global security level [11,26].

2.3 Hardware Sources

Transitions can occur in the hardware as soon as different data are successively manipulated. From a software point of view, the more visible case is one of the architectural registers or General Purpose Registers (GPRs). An executed application is directly able to know if it currently overwrites a destination register.

However, the issue is more important when it directly concerns the microarchitecture. Most of the potential transition leakage sources are completely abstract by the ISA: omit microarchitectural mechanisms can lead to security reduction [5]. By its structure, a processor pipeline unfortunately contributes to potential transitions: the same mechanisms process the different instructions cycle after cycle. Based on sets of microbenchmarks, different studies [23,12,27] expose the numerous potential leakage sources in current implementations. On both Arm [23,12] or RISC-V ISA [27] processors, multiple hardware mechanisms are then highlighted like pipeline registers, internal buffers or memory ports. By evaluating different boards, including SoCs and FPGAs with different ISAs, Marshall et al. have also highlighted that transition leakages really concern most of the systems [23]. Clearly, it appears that the microarchitecture has an important impact on leakage [3]. The issue is even deeper if we consider more complex microarchitectures like superscalar processors [7,19]. Indeed, the number of potential leaky mechanisms increases with the complexity, and the number of possible transitions is also higher due to simultaneous executions.

To deal with this complexity, an important part of the literature focused on building simulators to estimate power consumption efficiently for a given target [34,24,21,8,36] and then provide meaningful information for software developers about potential leakage. However, this type of simulator needs to be updated for each new microarchitecture to ensure that no source of leakage has been overlooked. An interesting approach is to automatically detect them at design time [30,4,18,6]. In this way, we ensure that results are automatically updated with any hardware changes.

2.4 Protection Strategies

Based on the detection of leakage sources, several strategies have been proposed to protect the system.

Pure software solutions try to catch the problem by only modifying the application itself. It leads to constraints for the software developer or compiler [19], which means they need information about the microarchitecture (*e.g.* pipeline stages). This is particularly useful to apply patches to existing hardware. However, it does not tackle the issue at its source and breaks the hardware abstraction from the ISA: the same program cannot be safely executed on different targets without changes. Because pure software solutions are difficult to efficiently implement, another approach is to directly modify the instruction set to add specific instructions. FENL [16] is a proposition of fencing to overwrite the different hardware registers and prevent transitions between the previous/next instructions. But this kind of approach involves modifying or recompiling the software and ensuring that the new instructions are efficiently introduced.

Another approach is a pure hardware solution where the microarchitecture is directly modified, allowing the code execution with transparent solutions. This principle is used in PARAM [4] to obfuscate direct value manipulation. However, they do not consider transition leakages. It can also be used to transparently add another layer of masking to increase the protection order [13]. But fundamentally, this approach does not remove the existence of transitions at their source. COCO [18] tackles this problem by splitting the hardening process between hardware and software. Deeply hidden transitions, such as unintended reads or switches in multiplexer trees, are effectively removed by modifying the hardware. But for a complete secured implementation, an important part is still delegated to the software, particularly to remove leakages due to successive instructions or architectural overwriting. This approach is interesting for simple processors like the IBEX core, whose data path is not pipelined. However, it is not enough when hidden microarchitectural buffers can keep sensitive data for multiple cycles.

For the rest of this paper, we propose a generic design methodology to tackle transition leakages by directly removing them from the microarchitecture. Our goal is to ensure that, independently of the software choices, the execution will not lead to a reduction in the security order due to the microarchitecture, even with pipelined or buffered mechanisms. Another kind of leakage targeting the hardware is glitches. However, they are due to physical phenomena considered at a lower abstraction level than the microarchitecture. In this paper, we only focus on the microarchitectural issue of transition leakages, trying to first eliminate this issue before integrating complementary protections during other design steps.

3 Methodology

To design a secure processor, one main challenge is to select a suitable abstraction level to make the hardening process efficient in terms of complexity and hardware overhead. While the ISA level does not allow a precise representation of the

system, consider directly all the logic gates leads to numerous specific changes. Hopefully, hardware description languages (HDLs) allow to efficiently describe a system from its microarchitecture specifications. Moreover, recent improvements now allow to efficiently produce RTL models while directly integrating generic strategies, such as hardening techniques.

In this work, we propose different hardening strategies at the microarchitecture level (described in Section 6). They allow different performance/security trade-offs, varying in the considered leakage sources (whole logic or registers only) and the execution throughput. In order to compare several processor configurations (number of pipeline stages, hardening strategies applied, *etc.*), we propose an iterative design methodology based on 4 steps. (a) From the target microarchitecture specifications, a RTL model of the target is produced. (b) Using a precise leakage model relying on the enumeration of all register and wire transitions, (c) leaky transitions are generated using a collection of software microbenchmarks suited for vulnerability assessment at the microarchitecture level. Then, with the help of a generic hardening approach, (d) the microarchitecture specifications are enriched to mitigate the diagnosed vulnerability depending on the performance overhead criteria.

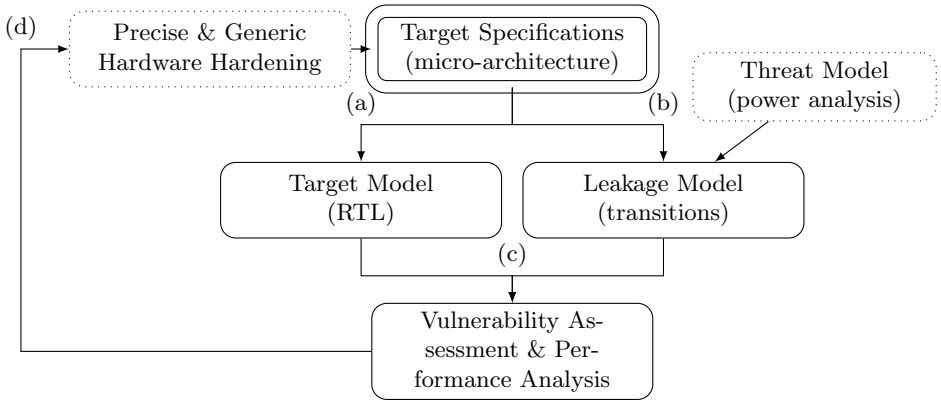


Fig. 1: Proposed methodology for generic and iterative microarchitecture hardening against power-based side-channel attacks.

In our case, to generalize the application of hardening strategies, we described in Section 4 how we increase this methodology by previously designing a set of fundamental blocks used to describe the microarchitectures. However, note that our methodology is still relevant for existing processors: the different hardening strategies must then be applied directly to each internal mechanism.

4 Targeted Design and Model Extraction

4.1 Generic Design Library

The goal of our methodology is to detail the needed steps to design a transition leakage-free processor. For that, some important changes in the whole microarchitecture will be needed to prevent these transitions. Instead of modifying each register or mechanism one by one, we decided to first design a generic design library. The idea is to define the fundamental modules before instantiating them to create the final design: register, buses with separated control and data parts, FIFO, *etc.* Then, to apply the design strategies described in the following Section 6, we only need to modify once the generic modules.

To push this strategy to the limit, we decided to use the Chisel language [2]. Based on the Scala language, it allows to efficiently generate Verilog/SystemVerilog descriptions and stay at the RTL level while benefiting from modern language features. Particularly, we use object-oriented programming and inheritance to design highly generic modules. For example, it is the same custom register module which will be used in the whole design, in the processor pipeline as well as in the memory controller. The whole code is available online [14].

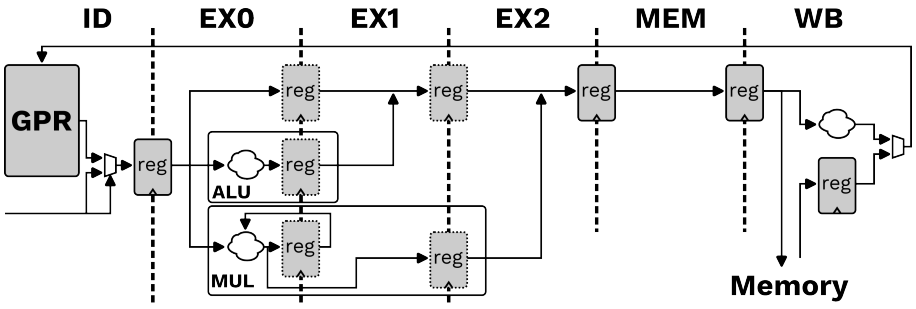


Fig. 2: Datapath of the targeted processor microarchitecture.

4.2 Microarchitecture description

Using these generic modules, we designed the processor whose datapath is described in the Figure 2. Based on the RV32I ISA [35], it also supports different extensions like M (multiply and divide) or ZiCsr.

Its microarchitecture is based on a configurable pipeline with four to seven stages. Each register (in gray), excepted GPRs, is based on the previously defined generic register module. The dotted ones on the Figure 2 are registers which can be enabled or disabled during SystemVerilog generation. In the next sections, this will be useful to evaluate that our methodology can be applied on design with microarchitectural variations.

At the microarchitecture level, design choices can directly impact how the values in registers are modified. In our case, we have to notice that registers are updated only when a new valid instruction is in the corresponding stage/unit. This is a common practice to reduce consumption due to bit switches. However, it leads to the observation that a value is still in a register after many cycles, which can lead to hidden data transitions.

For the rest of this paper, we identify two different microarchitectures. C5U (*Core 5 Unprotected*) is the simple version with a five-stage pipeline: execution (EX) - memory (MEM) - write back (WB) stages are implemented. C7U (*Core 7 Unprotected*) is the most complex version with a seven-stage pipeline: EX is split into three stages and all the execution units are pipelined (one result register).

4.3 Simulation Analysis

The first step of the methodology involves generating an accurate simulation model from microarchitectural sources/description of our design to analyze potential leaky transitions. After generating the SystemVerilog description, this is done by compiling a fast executable to simulate our design using Verilator [32] able to generate *.vcd* file. This format is the standard to represent digital design waveforms by simply indicating value changes in each cycle. In our case, this is the perfect information to track potential transitions: this can simply be done by parsing files (*e.g.* with a Python 3 script). Moreover, it is fast applicable in any hardware design, independently of the internal changes.

5 Microbenchmark

After the extraction of the simulation model, the second step of our methodology is to try to generate transitions to detect microarchitectural leakage sources. A common strategy in the literature [23,12,27] is based on microbenchmarks using specific instruction patterns. If MicroPlumber [27] also uses the RISC-V ISA, it only targets a simple PicoRV32 core. Based on microbenchmarks available for the ARM ISA [23,12], we developed our own set for the RISC-V ISA. They are also available in an online repository [14].

5.1 Test Principle

Listing 1.1: Microbenchmark test structure

```
1  jal trigger_on
2  INSERT_NOP
3  instr0  # Handle op0
4  instr1  # Handle op1
5  instr2  # Handle op2
6  instr3  # Handle op3
7  INSERT_NOP
8  jal trigger_off
```

The idea behind microbenchmarking is to design a set of simple program patterns to test different parts of the microarchitecture. In our case, we want to reproduce transitions in any part of our processor. In a white-box scenario with a defined microarchitecture, this is possible with only a small set of microbenchmarks. Instructions could be considered in subsets using the same hardware mechanisms: ALU, multiply, branch and memories. It then considerably reduces the number of necessary tests.

All our microbenchmarks are designed following the structure described on Listing 1.1. First, a trigger is launched at the beginning (`jal trigger_on`) and stopped at the end (`jal trigger_off`) of each test. It allows synchronizing measurements and the different analyses. Then, `nop` instructions are inserted (here using a macro `INSERT_NOP`) to isolate the targeted pattern. Finally, a sequence of instructions is used to manipulate data and potentially generate transitions.

While two subsequent instructions are sufficient to produce a transition, microbenchmarks presented here are also adapted for complex microarchitectures with multiple buffers, FIFOs or superscalar execution. In these systems, additional instructions are needed to cover all the cases (fill the buffer, evaluate parallel operations, *etc.*). In the case of pipelined designs such as our target datapath presented in Figure 2, two subsequent instructions are always enough to generate any transition. However, we decided to keep tracking of the four operands to highlight the detected leakage and their behaviors. It leads to 6 possible transitions which must be tracked.

5.2 Test Examples

We now describe some of the tests we used to generate transitions in our microarchitecture. Each time, the expected results have been verified by directly inspecting simulated execution traces: this is a benefit to directly have access to the RTL description. Another one is to be able to directly check if all the microarchitectural parts have been tested.

Listing 1.2: `alu_alu_gpr`

```
1 xor t0, zero, op0
2 xor t0, op1, zero
3 xor t0, zero, op2
4 xor t0, op3, zero
```

Listing 1.3: `alu_alu_res`

```
1 xor t0, zero, op0
2 xor t1, op1, zero
3 xor zero, zero, op2
4 xor zero, op3, zero
```

Listing 1.2 and Listing 1.3 are tests focusing on leakage due to arithmetic and logic sequences. In `alu_alu_gpr`, the goal is to generate transitions in all the microarchitectural registers and wires where the instruction results are stored, but also in an architectural register, here `t0`. In `alu_alu_res`, the goal is almost the same, excepting that no transition is expected in `t0`. Then, by executing both tests, we are supposed to distinguish the impact of architectural and microarchitectural registers.

Listing 1.4: `mul_mul_s1`

```
1 li t4, -1
2 mulhu t0, op0, t4
3 mulhu t1, op1, t4
4 mulhu zero, op2, t4
5 mulhu zero, op3, t4
```

Listing 1.5: `mul_mul_res`

```
1 li t4, 1
2 mul t0, t4, op0
3 mul t1, op1, t4
4 mul zero, t4, op2
5 mul zero, op3, t4
```

Listing 1.4 and Listing 1.5 focus on leakages due to multiplication support in the microarchitecture. Generally, multipliers are implemented in dedicated parts of the system. Then, the corresponding wires and registers are used only when multiplication instructions are executed. Here, `mul_mul_s1` focuses on possible transitions between the first operand of the multiplication and `mul_mul_res` between the results.

Listing 1.6: `leq_lw_lw`

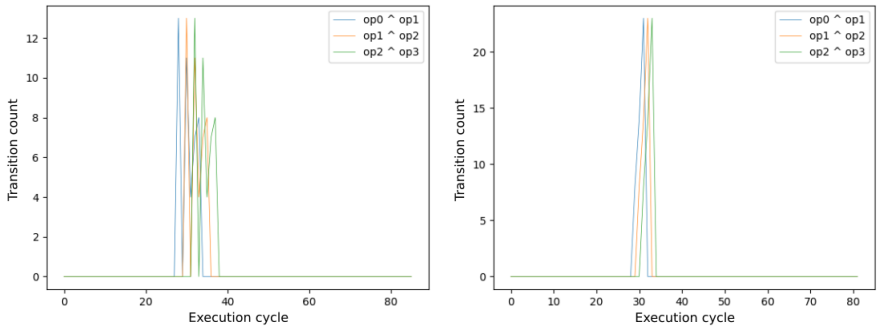
```
1 lw t0, 0(t4)
2 lw t1, 4(t4)
3 lw t2, 8(t4)
4 lw t3, 12(t4)
```

Listing 1.7: `aeq_sw_sw`

```
1 sw t0, 0(t4)
2 sw t1, 0(t4)
3 sw t2, 0(t4)
4 sw t3, 0(t4)
```

Transitions can occur in the whole processor pipeline, but also in any hardware mechanism where data are manipulated, such as memories. Some of our microbenchmarks try to evaluate them using load and store instructions. On Listing 1.6, `leq_lw_lw` allows to evaluate leakage when multiple sensitive operands are successively executed. On Listing 1.7, `aeq_sw_sw` is used to detect the impact of transition when sensitive values are overwriting each other in memory.

All the microbenchmarks previously described are not enough to test all the possible data transitions in the microarchitecture. Then, we also provide tests to target other mechanisms, such as the first instruction operand registers, the second operand registers, the resize buffer used by `load` instructions, the forwarding logic and the GPR read ports with immediate/register decoding.



(a) Transition count for `alu_alu_gpr` (b) Transition count for `alu_alu_res`

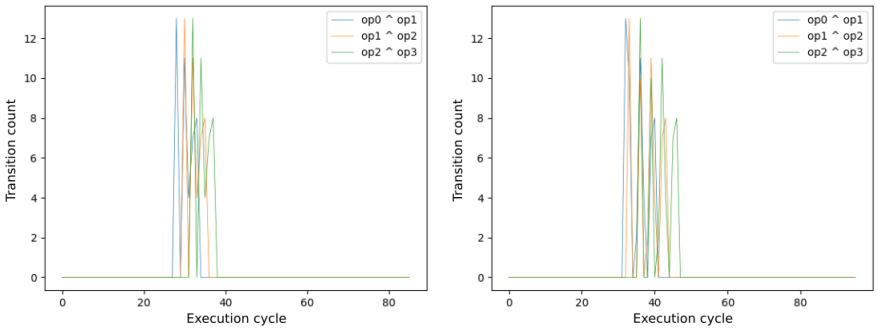
Fig. 3: Analysis of ALU operations on C5U.

5.3 Model Analysis

After defining the different microbenchmarks, we can execute them, generate the representing *.vcd* files for each core and perform the analysis to detect the number of transitions and the responsible wires/registers. In our case, the execution of the microbenchmark on our cores is completely deterministic: only one execution of each test is enough to know in simulation if a transition occurs or not. Then, to perform the analysis, we simply fix the values of $op0$ to $op3$ and directly track them in the *.vcd* files. In our case, it only takes a few dozen minutes from the hardware generation to the analysis end on a desktop computer.

Figure 3a and Figure 3b show the number of transitions occurring when executing respectively `alu_alu_gpr` and `alu_alu_res` on C5U. This analysis is simply performed by parsing the generated *.vcd*: for each cycle, we compare if a transition occurs between known operand values (from $op0$ to $op3$). Finally, we are able to know for each microbenchmark if a transition occurs, at which cycle and on which signal.

As expected, we see the transitions between the operands occurring successively during a few cycles. In the case of `alu_alu_gpr`, because these traces are completely noise-free, we are even able to distinguish subtle variations at the end of the curves, representing the last transition during the write-back.



(a) Transition count for C5U (b) Transition count for C7U

Fig. 4: Analysis of the `mul_mul_s1` microbenchmark.

Figure 4a and Figure 4b show the number of transitions occurring when executing `mul_mul_s1` on both cores C5U and C7U. We can directly observe the impact of adding the microarchitecture by adding extra-registers in C7U.

Finally, for C5U and C7U, it is respectively 194 and 261 SystemVerilog wires or registers (`logic`) with transitions which have been detected in the pipeline and the internal RAM. If most of them are obviously directly dependent on this level, it highlights the importance of a generic approach to secure the system and ensure that no mechanism is missed.

6 Root Cause Protection Principles

After detecting leakage sources, secure the design involves modifying the microarchitecture. Particularly, we need a way to prevent all the possible leaky transitions. To keep our methodology as generic as possible, we need to define protection strategies, which can be applied to the whole system. For that purpose, we decided to generalize a principle called *data overwriting*.

6.1 Data Overwriting Principle

We saw in Section 2 that transitions occur when signal or register values directly change from a sensitive value to another. This type of event occurs regularly in processors where operations follow one another cycle by cycle. Some work [16] have shown that this can be prevented by overwriting hardware between two data manipulation. Fundamentally, the goal is to transform any possible $op0 \rightarrow op1$ transition (with $op1$ and $op2$ sensitive values) to $op0 \rightarrow tmp \rightarrow op1$ with tmp a temporary independent value.

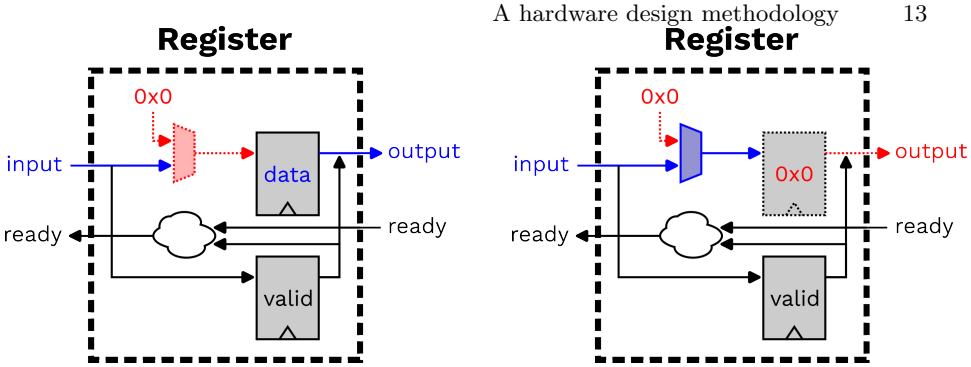
Theoretically, data overwriting is possible at different levels. First at the software-level, we can possibly insert extra instructions to push some zero or random values into hardware mechanisms. However, due to the abstraction from the ISA, it becomes difficult to put into practice: any missed microarchitectural register can lead to a leakage. Then, another possibility is to directly enhance the ISA with dedicated instructions to clear the microarchitecture [16]. Called fencing, this strategy requires recompiling the executed programs to select when a clear is needed. Moreover, fencing has a major limitation: it cannot be similarly applied to architectural registers (*e.g.* GPRs) where data must not be modified. Finally, an ideal case would be to directly prevent the appearance of transitions in the microarchitecture at the hardware level. Then, any program can be safely executed, without necessary modifications during compilation. The challenge is then to ensure that no source of leakage is overlooked.

For that, we define five complementary strategies to systematically apply overwriting at the hardware-level. We then integrate them into our generic bricks defined with the Chisel language and described in Subsection 4.1, allowing application to the entire design. Then, using iterations of our methodology, we ensure that there is no omitted leakage source.

6.2 Strategy 1.a: Complete Stage Overwriting

The first strategy called *complete stage overwriting* is described on the Figure 5. It is the direct implementation of the overwriting principle.

Each cycle, a validity bit allows to indicate if the corresponding register holds a data (Figure 5a). When the data is consumed by the following pipeline stage (indicated with a ready signal), an extra-cycle is used to overwrite the register with a zero (`0x0`) value. Finally, it is only after this operation that the register can be reused to hold a new data: it ensures that the transition $op0 \rightarrow op1$ is transformed into $op0 \rightarrow \texttt{0x0} \rightarrow op1$.



(a) Cycle N (b) Cycle N+1

Fig. 5: Register overwriting

An essential point to note here is that for each overwriting operation performed in the register, the zero value is also propagated through the output signal. This behavior allows to also initialize all combinatorial signals which depend on this register. Finally, by adding this extra overwriting cycle, it is not only the registers which are initialized, but the complete pipeline stage including the combinatorial logic.

6.3 Strategy 1.b: Delayed Data Multiplexing

Execution in a complete system based on a processor is not always linear. For example, multiplexers in the microarchitecture allow to select data from multiple sources: register forwarding, skid buffers, *etc.* In these cases, complete stage overwriting is not sufficient for the multiplexer output if the sources are not always synchronized (overwritten at the same cycle).

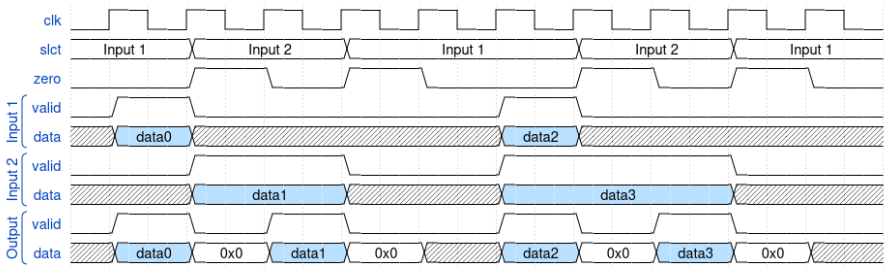


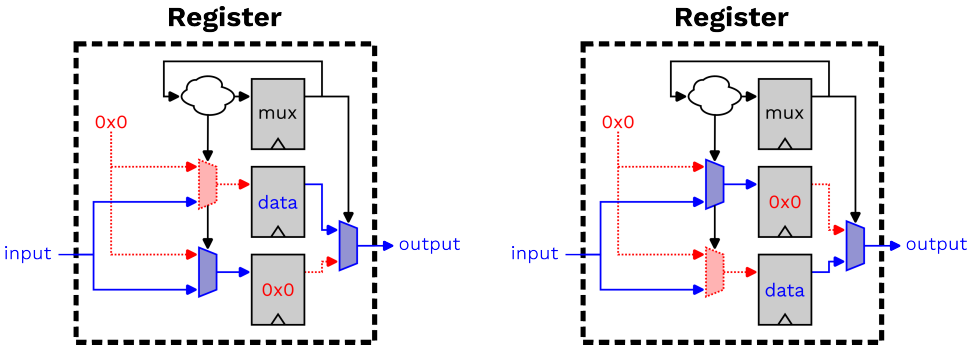
Fig. 6: Delayed data multiplexing description.

For these mechanisms, we implemented a complementary generic strategy called *delayed data multiplexing* described on the Figure 6. The main idea is simply to introduce an overwrite cycle after each valid multiplexer output.

6.4 Strategy 2: Microarchitectural Register Duplication

Complete stage overwriting combined with delayed data multiplexing allow protecting all the transition sources in a pipelined structure. However, they have a major drawback for performances by inserting extra cycles. In the worst case, we can estimate that it can divide by two the execution rate: one extra overwriting cycle for each real operation cycle. Moreover, as shown in the Section 2, most of the leakages are caused by transitions directly occurring in registers. Depending on the constraints, the strategies 1.a and 1.b can be excessive.

For that purpose, we decided to evaluate another generic strategy called *microarchitectural register duplication*. The main idea is to parallelize the register overwriting and the real operation write to prevent the extra cycle. For that purpose, we implement the mechanism described on the Figure 7.



(a) Cycle N (b) Cycle N+1

Fig. 7: Register duplication

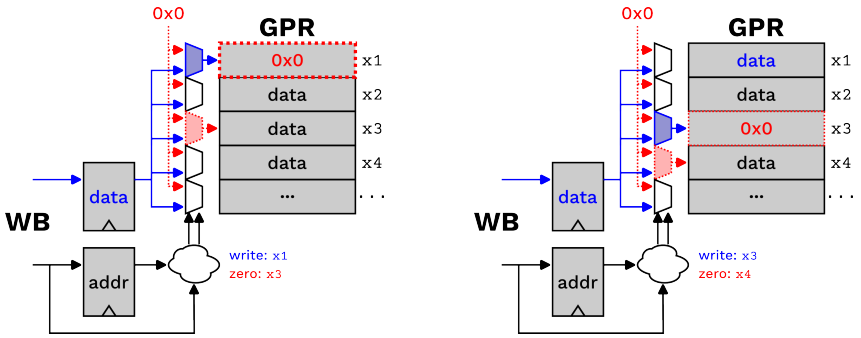
All the data registers are duplicated. In the cycle N described on the Figure 7a, one part contains a real data and the other the zero value. A new multiplexer allows selecting them: the data register is selected for the output but the zero register is selected for the input. If the data is read and a new one is written, then the roles are reversed in the next cycle described on the Figure 7b. Finally, the transition $op0 \rightarrow op1$ is split into two transitions $op0 \rightarrow$ `0x0` and `0x0` $\rightarrow op1$.

The initial idea here is similar to the principle of Secure Double Rate Registers [9]. However, we have two implementation differences. First, the registers are here parallelized (not pipelined): the latency is not impacted in our case. Introducing new stages like with Secure Double Rate Registers [9] can have a critical impact in the case of processor pipelines with data dependencies or branch management. Second, in our case, we do not need to double the whole frequency to have a similar rate: it is completely transparent as long as the critical path is not impacted.

6.5 Strategy 3.a: Architectural Register Pre-Writing

Previous strategies can be applied to any microarchitectural registers to prevent transitions. However, architectural registers (the ones defined by the ISA such as GPRs) must also be protected if we want to protect any software execution without specific changes. For that purpose, we defined another strategy called *register pre-writing*.

Unlike microarchitectural registers, an architectural register cannot simply be overwritten after usage: a data stored inside must remain as long as another valid data is written. This can lead to a leaky transition when the old and new data are both sensitive. Moreover, simply duplicate registers is here not a valid solution: the RV32I ISA considers 31 32-bit registers (`x0` is hardwired to 0). Then, we implemented a new mechanism described on Figure 8.



(a) Cycle N (b) Cycle N+1

Fig. 8: GPR overwriting

The main principle of this strategy is to ensure that each register is pre-written by an insensitive value (here `0x0`) just before the new valid data. For that purpose, we add some logic and registers (where strategies 1 or 2 can be applied) to each write port of our GPRs. Their goal is to manage the pre-writing and hold the information to perform the real write in the next cycle. On the Figure 8a, we can see that the previously pre-written register `x1` is receiving a new data while `x3` is not yet ready. In the next cycle on the Figure 8b, it is finally another real data which is written in `x3` when is pre-written.

Finally, for each write operation, there is only one cycle where no valid data is present on a GPR. However, during this time, the next data is available in the new write port register. Using a mechanism similar to a register forwarding, it is then still possible to transfer each time a valid data to a read port: if the read address is the same as the one in the write port register, the data is directly transferred, else the GPR is read. The architectural state is still preserved.

6.6 Strategy 3.b: Decoupled Memory Operations

Finally, the last hardware mechanism where data transitions can occur is memories. Basically, memories are part of the architectural state: the first three strate-

gies cannot be strictly applied. Moreover, simultaneous write/pre-write of the fourth strategy needs two simultaneous write ports which are not always available for memories. In this case, we decided to establish a fifth strategy based on Finite State Machines (FSMs) to perform *decoupled memory operations*.

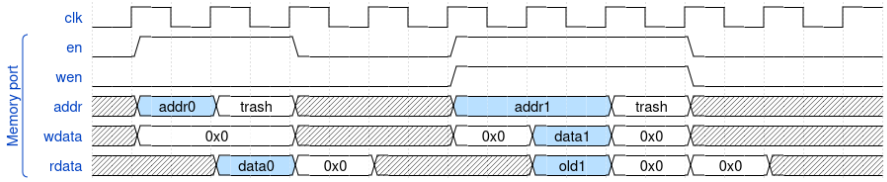


Fig. 9: New memory operation description with a real port.

In the case of read memory operations, transitions must be prevented between successive reads. For that, we decide to implement two-cycle read operations as described on the Figure 9 (first transaction). Each real read is followed by an extra-cycle where a read to another address is performed. However, this other address must be carefully selected and ensure that no leaky data is inside. For that, we decided to lock the last four bytes of each memory for hardware operation: the software is not supposed to use them. Then, these memory locations can be used as the default trash addresses, like the `x0` register in the GPRs. In this way, no transition is possible on the `rdata` signal and the `wdata` is set to 0 because it is unused.

The case of write operations is a bit trickier. First, like for an architectural register write, a pre-write is needed to ensure that no transition occurs between the old and the new data. This is the first part of the second transaction on the Figure 9. Then, the real write can be performed. After that, we must ensure that no transaction can occur with a future operation on both `wdata` or `rdata`. Like for the read operation, an extra-cycle is then needed to perform a write to the trash address.

Finally, decoupled memory operations can also have an important impact on performance by slowing memory access. However, they must be considered when coupled with the other strategies presented here. With complete stage overwriting, extra-cycles are already introduced: the supplement impact will be here limited. In the case of register duplication, we only consider leakage occurring in the register. Then, transactions presented on the Figure 9 can be simplified: only the pre-write is needed.

6.7 Implementations

For the evaluation in the next section, we have to compare the different strategies on different microarchitectures. It results in the implementations presented on the Table 1.

Each core is a different version of the configurable datapath presented on the Figure 2. *C5** cores use the simplest version of the datapath where no execution unit or intermediate EX registers are used. *C7** cores use the complete version

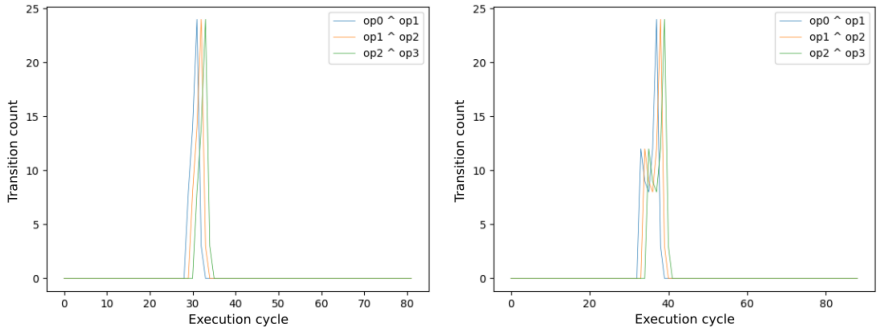
Table 1: List of the different processor configurations.

Configuration	Datapath stages	EX unit registers	S1.a	S1.b	S2	S3.a	S3.b
C5U	4	No	No	No	No	No	No
C5S1	4	No	Yes	Yes	No	Yes	Yes
C5S2	4	No	No	No	Yes	Yes	Partial
C7U	6	Yes	No	No	No	No	No
C7S1	6	Yes	Yes	Yes	No	Yes	Yes
C7S2	6	Yes	No	No	Yes	Yes	Partial

of the datapath, with all the registers enabled. **U* cores are the unprotected ones. **S1* and **S2* integrate respectively different strategies.

7 Evaluation

After implementing the different protection strategies, it is necessary to re-evaluate the targets to verify the efficiency: it is the last step of our methodology. For that, like in a classic design flow, we first perform verification in simulations using the same tools as previously. The results presented in this section were obtained after several iterations of our methodology, ensuring that no leakage source has been overlooked at the RTL level. Then, we validate the whole results by performing measurements and analysis on a real system.



(a) Transition count for C5S2 (b) Transition count for C7S2

Fig. 10: Analysis of the `alu_alu_gpr` microbenchmark.

7.1 Simulation model

In the same way as in the Section 5, we execute once all the microbenchmarks on each target with fixed operands to allow direct transition detection with *.vcd* file analysis.

In the case of complete stage overwriting implementations, no more transitions were detected in the processor pipeline as well as in the memory controller. If it is the final expected result, a particular challenge was not to forget any leakage sources. Particularly, if the use of Chisel generic modules was enough for register modification (which are a very large proportion of the leakage sources), the implementations of delayed data multiplexing were done manually. It highlights the interest in a methodology with both a global approach combined with iterations to manage particular cases.

On the other side, register duplication is easier to implement: generic modules are enough because only the registers are considered here. Then, as expected, transitions can still be detected in wires, as described in the Figure 10. As before, we can even see that the transitions are still implementation-dependent with differences between *C5S2* and *C7S2*.

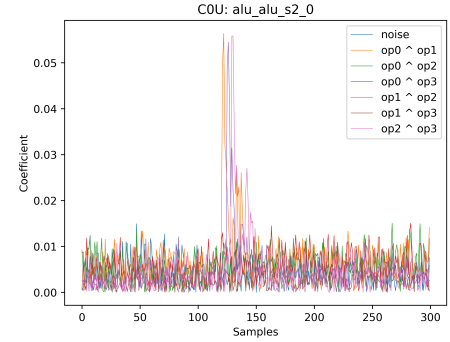
7.2 Real Measurements

Simply analyzing transitions from *.vcd* is not enough to evaluate the efficiency of protection strategies in the real world. Some steps, such as logic synthesis or the place-and-route, can influence the behavior of mechanisms previously defined at the microarchitectural-level. Then, we perform a two-step security assessment, with evaluations on real FPGA targets to confirm the previous simulated results.

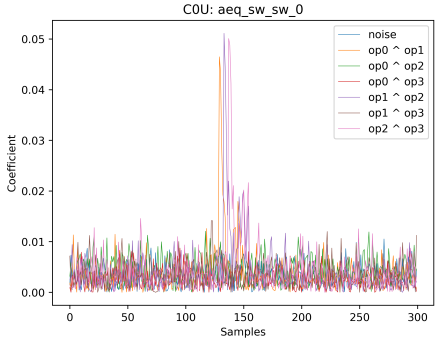
For that, we implemented our cores on the ChipWhisperer platform [25] with the CW305 FPGA target, integrating a Xilinx Artix-7. The processor is configured to run at 25 MHz when the sampling frequency is 100 MHz (4 samples by cycle). Each microbenchmark is then executed 50.000 times with random input operands, allowing to perform Correlation Power Analysis (CPA). For each execution on *U and *S2 cores, 300 samples are performed. Due to the insertion of extra cycles in *S1 cores, more cycles are needed to execute the different programs: we use for them 500 samples.

The Figure 12 (right part) describes the CPA results for the `aeq_sw_sw` microbenchmark executed on the three versions of the C5 core. In each figure, the correlation for different transitions is calculated with valid operand values, but also with random values to estimate the noise range. With C5U, clear correlation peaks can be detected after $\tilde{1}40$ samples due to successive manipulation of the operands. On the other hand, no clear correlation is detected between the power consumption and data transitions with the C5S1 and C5S2 cores. Then, in this case, both secure implementations seem to offer the same protection.

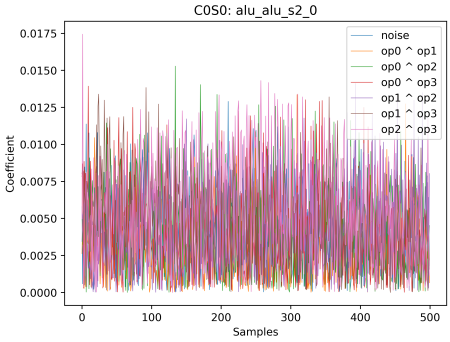
The Figure 12 (left part) describes the CPA results for the `alu_alu_s2` microbenchmark (leakages due to the second operand in arithmetic and logic operations). Like for the previous microbenchmark, we can see in Figure 11a and Figure 11c that C5U and C5S1 are still respectively leaky and non-leaky designs. The main information here is in Figure 11e where we can observe weak correlation peaks. Then, for this microbenchmark, register duplication reduces the leakages (correlation decreases from 0.55 to 0.02), but it does not eliminate them completely.



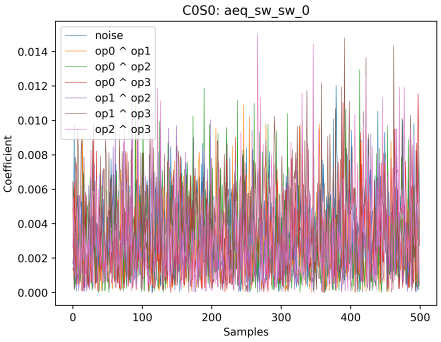
(a) CPA for C5U



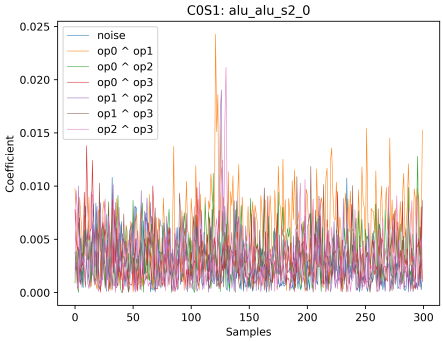
(b) CPA for C5U



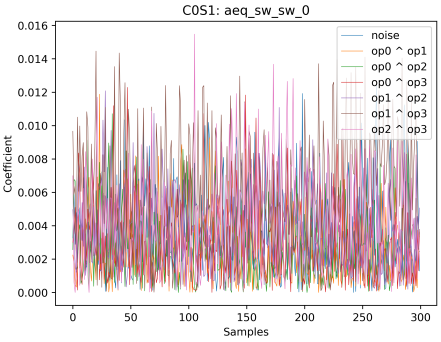
(c) CPA for C5S1



(d) CPA for C5S1



(e) CPA for C5S2



(f) CPA for C5S2

Fig. 11: CPA for the `aeq_sw_sw` microbenchmark.

Fig. 12: CPA for the `alu_alu_s2` (left) and `aeq_sw_sw` (right) microbenchmarks.

Finally, considering that we obtained similar results with the other microbenchmarks (C5S2 is leaky depending on the targeted hardware mechanism), we can conclude that the observations on the FPGA target confirm our previous results in the simulations. As expected, completely removes transition at the RTL level allows to remove the root cause of transition leakage. On the other hand, only preventing transition in registers is not sufficient: leakages are deduced but still exist, and can be detected in some cases.

7.3 Overhead

The different strategies described in the Section 6 deeply impact the initial microarchitectures. However, in addition to security, processors must often satisfy other constraints like performance or resource utilization. The different results are summarized in the Table 2.

Table 2: Overhead of the different strategies.

Configuration	Embench (cycle)	Embench (ratio)	LUT (count)	LUT (ratio)	FF (count)	FF (ratio)
C5U	4 412 169	N/A	5 027	N/A	2 498	N/A
C5S1	8 222 197	+86.35%	5 295	+05.33%	2 520	+00.88%
C5S2	4 529 543	+02.66%	6 910	+37.46%	3 102	+24.18%
C7U	5 432 114	N/A	5 617	N/A	3 156	N/A
C7S1	8 051 161	+48.21%	6 244	+11.16%	3 224	+02.15%
C7S2	5 433 002	+00.02%	7.162	+27.51%	4.211	+33.43%

Performance To evaluate performance, we executed the Embench [1] benchmark on all our microarchitectures and we averaged the number of cycles. Globally, the obtained results correspond to our expectations. Implementations with complete stage overwriting and associated strategies (**S1*) are highly impacted when changes to register duplication strategies (**S2*) are negligible. This is mostly due to the addition of the extra-cycle. However, we can see that this impact is highly reduced with the second core version (from +86.35% with C5S1 to +48.21% with C7S1). This is mainly due to the impact of false branch predictions, which already have an important impact on the reference version (C5U).

We choose here not to present the impact on the critical path (and the frequency) of our strategies due to not relevant results: the initial targeted microarchitectures have not been particularly optimized to increase the frequency. Then, the impact of simple changes (*e.g.* simply add a multiplexer) will not be as representative as for a state-of-the-art processor designed solely for performance.

Resource utilization To evaluate resource utilization, we decided to reuse results from Vivado 2023.1 during the implementation on the CW305 target. Each presented results are only for the processor pipeline, excluding the memories, the peripherals or the branch prediction mechanisms, which are major resource consumers in the initial designs. Both the number of used LUTs and registers

(flip-flops) are presented. Here again, the obtained results correspond to our expectations. **S2* implementations use more resources due to register duplication. Depending on the implementations, the register increase is evaluated to +24.18% or +33.43%: it is more important for *C7S2* due to the higher number of pipeline stages.

8 Conclusion

Power consumption measurements represent a threat to recover information from hardware systems. Particularly, data transitions are a burden to implement secure software algorithms.

In this paper, we establish a methodology to design transition leakage free processors. From a RTL model, we explained how we extract at simulation time information about wire and register states for each cycle. Combined with microbenchmarks, we use this step to precisely analyze and detect leakage sources in the microarchitecture. Then, we propose five generic strategies to tackle the root cause of these leakages: the transitions themselves. Using generic blocks defined with the Chisel language, we modify the microarchitecture of two variants of a custom RISC-V processor. Finally, we confirm the efficiency of our strategy by executing the microbenchmarks on an FPGA target and by performing real measurements. The different results highlight a clear trade-off between security and performances. If complete stage overwriting effectively removes all potential leakage sources, it has an important impact on performances (+86.35% number of cycles in the worst-case). On the other side, register duplication keeps transition in wires which can possibly be detected with better experimentation setup.

Future work will concern the other system layers to be able to consider security issues as a whole. Higher layers such as the ISA will be explored to allow a better adaptation of the system to the needs of the applications. Instead of fence instructions which need to be inserted at specific places depending on the microarchitecture to be effective, another promising strategy is contextualization to secure whole sensitive code blocks. Depending on the constraints of the targeted application, switching between the different strategies presented here can be an interesting trade-off. Lower layers will also be considered to ensure that no leakage sources are inserted later in the design process (during synthesis or place-and-route). This is directly linked to the management of glitches, which can be introduced in lower levels of the design: each security issue must be tackled at the corresponding system layer.

References

1. Embench: A Modern Embedded Benchmark Suite (2021), `https://www.embench.org/`
2. Chisel/FIRRTL Hardware Compiler Framework (2023), `https://www.chisel-lang.org/`

3. Arora, V., Buhan, I., Perin, G., Picek, S.: A Tale of Two Boards: On the Influence of Microarchitecture on Side-Channel Leakage. In: Grosso, V., Pöppelmann, T. (eds.) Smart Card Research and Advanced Applications. vol. 13173, pp. 80–96. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-97348-3_5, https://link.springer.com/10.1007/978-3-030-97348-3_5
4. Arsath K F, M., Ganesan, V., Bodduna, R., Rebeiro, C.: PARAM: A Microprocessor Hardened for Power Side-Channel Attack Resistance. In: 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 23–34 (Dec 2020). https://doi.org/10.1109/HOST45689.2020.9300263
5. Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., Standaert, F.X.: On the Cost of Lazy Engineering for Masked Software Implementations. In: Joye, M., Moradi, A. (eds.) Smart Card Research and Advanced Applications. pp. 64–81. Lecture Notes in Computer Science, Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-16763-3_5
6. Barenghi, A., Brevi, M., Fornaciari, W., Pelosi, G., Zoni, D.: Integrating Side Channel Security in the FPGA Hardware Design Flow. In: Bertoni, G.M., Regazzoni, F. (eds.) Constructive Side-Channel Analysis and Secure Design. vol. 12244, pp. 275–290. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-68773-1_13, http://link.springer.com/10.1007/978-3-030-68773-1_13
7. Barenghi, A., Pelosi, G.: Side-Channel Security of Superscalar CPUs: Evaluating the Impact of Micro-Architectural Features. In: Proceedings of the 55th Annual Design Automation Conference. pp. 1–6. ACM, San Francisco California (Jun 2018). https://doi.org/10.1145/3195970.3196112, https://dl.acm.org/doi/10.1145/3195970.3196112
8. Barthe, G., Gourjon, M., Grégoire, B., Orlt, M., Paglialonga, C., Porth, L.: Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 189–228 (Feb 2021). https://doi.org/10.46586/tches.v2021.i2.189-228, https://tches.iacr.org/index.php/TCHES/article/view/8792
9. Bellizia, D., Bongiovanni, S., Monsurrò, P., Scotti, G., Trifiletti, A., Trotta, F.B.: Secure Double Rate Registers as an RTL Countermeasure Against Power Analysis Attacks. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **26**(7), 1368–1376 (Jul 2018). https://doi.org/10.1109/TVLSI.2018.2816914, https://ieeexplore.ieee.org/document/8327903
10. Bos, J.W., Gourjon, M., Renes, J., Schneider, T., van Vredendaal, C.: Masking Kyber: First- and Higher-Order Implementations. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 173–214 (Aug 2021). https://doi.org/10.46586/tches.v2021.i4.173-214, https://tches.iacr.org/index.php/TCHES/article/view/9064
11. Coron, J.S., Giraud, C., Prouff, E., Renner, S., Rivain, M., Vadnala, P.K.: Conversion of Security Proofs from One Leakage Model to Another: A New Issue. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Schindler, W., Huss, S.A. (eds.) Constructive Side-Channel Analysis and Secure Design. vol. 7275, pp. 69–81. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29912-4_6, http://link.springer.com/10.1007/978-3-642-29912-4_6

12. de Grandmaison, A., Heydemann, K., Meunier, Q.L.: ARMISTICE: Micro-Architectural Leakage Modelling for Masked Software Formal Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **41**(11), 3733–3744 (2022)
13. De Mulder, E., Gummalla, S., Hutter, M.: Protecting RISC-V against Side-Channel Attacks. In: 2019 56th ACM/IEEE Design Automation Conference (DAC). pp. 1–4. ACM, Las Vegas NV USA (Jun 2019). https://doi.org/10.1145/3316781.3323485, https://dl.acm.org/doi/10.1145/3316781.3323485
14. Escouteloup, M., Migliore, V.: Design platform and microbenchmarks (2025), https://gitlab.com/herd-ware/root
15. Gao, S., Großschädl, J., Marshall, B., Page, D., Pham, T., Regazzoni, F.: An Instruction Set Extension to Support Software-Based Masking. IACR Transactions on Cryptographic Hardware and Embedded Systems p. 43 (2020)
16. Gao, S., Marshall, B., Page, D., Pham, T.: FENL: An ISE to Mitigate Analogue Micro-Architectural Leakage. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 73–98 (Mar 2020). https://doi.org/10.13154/tches.v2020.i2.73-98, https://tches.iacr.org/index.php/TCHES/article/view/8545
17. Gao, S., Oswald, E., Page, D.: Reverse engineering the micro-architectural leakage features of a commercial processor. Cryptology ePrint Archive (2021)
18. Gigerl, B., Hadzic, V., Primas, R., Mangard, S., Bloem, R.: COCO: Co-Design and Co-Verification of Masked Software Implementations on CPUs. In: 30th USENIX Security Symposium (USENIX Security'21). p. 18 (2021)
19. Gigerl, B., Primas, R., Mangard, S.: Secure and Efficient Software Masking on Superscalar Pipelined Processors. In: Tibouchi, M., Wang, H. (eds.) Advances in Cryptology – ASIACRYPT 2021, vol. 13091, pp. 3–32. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-92075-3_1
20. Ishai, Y., Sahai, A., Wagner, D.: Private Circuits: Securing Hardware against Probing Attacks. In: Goos, G., Hartmanis, J., van Leeuwen, J., Boneh, D. (eds.) Advances in Cryptology - CRYPTO 2003. vol. 2729, pp. 463–481. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_27, http://link.springer.com/10.1007/978-3-540-45146-4_27
21. Le Corre, Y., Großschädl, J., Dinu, D.: Micro-Architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors. In: Fan, J., Gierlichs, B. (eds.) Constructive Side-Channel Analysis and Secure Design, vol. 10815, pp. 82–98. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-89641-0_5
22. Marshall, B., Page, D.: SME: Scalable Masking Extensions. Cryptology ePrint Archive p. 25 (2021)
23. Marshall, B., Page, D., Webb, J.: MIRACLE: MIcRo-ArChitectural Leakage Evaluation: A Study of Micro-Architectural Power Leakage across Many Devices. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 175–220 (Nov 2021). https://doi.org/10.46586/tches.v2022.i1.175-220
24. McCann, D., Whitnall, C., Oswald, E.: ELMO: Emulating Leaks for the ARM Cortex-M0 without Access to a Side Channel Lab. IACR Cryptol. ePrint Arch. **2016**, 517 (2016)
25. O'Flynn, C., Chen, Z.: Chipwhisperer: An open-source platform for hardware embedded security research. In: Constructive Side-Channel Analysis and Secure Design (COSADE). pp. 243–260. Springer, Paris, France (Apr 2014)

26. Papagiannopoulos, K., Veshchikov, N.: Mind the Gap: Towards Secure 1st-Order Masking in Software. In: Guilley, S. (ed.) Constructive Side-Channel Analysis and Secure Design, vol. 10348, pp. 282–297. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-64647-3_17
27. Roy, A., Schaumont, P.: Microplumber: Finding Hidden Sources of Power-Based SCL in Microcontrollers. In: 2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). pp. 762–765. IEEE, Knoxville, TN, USA (Jul 2024). https://doi.org/10.1109/ISVLSI61997.2024.00148, https://ieeexplore.ieee.org/document/10682629/
28. Shelton, M.A., Chmielewski, Ł., Samwel, N., Wagner, M., Batina, L., Yarom, Y.: Rosita++: Automatic Higher-Order Leakage Elimination from Cryptographic Code. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 685–699. ACM, Virtual Event Republic of Korea (Nov 2021). https://doi.org/10.1145/3460120.3485380, https://dl.acm.org/doi/10.1145/3460120.3485380
29. Shelton, M.A., Samwel, N., Batina, L., Regazzoni, F., Wagner, M., Yarom, Y.: Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. In: Proceedings 2021 Network and Distributed System Security Symposium. Internet Society, Virtual (2021). https://doi.org/10.14722/ndss.2021.23137
30. Šijačić, D., Balasch, J., Yang, B., Ghosh, S., Verbauwhede, I.: Towards Efficient and Automated Side Channel Evaluations at Design Time. Journal of Cryptographic Engineering **10**(4), 305–319 (Nov 2020). https://doi.org/10.1007/s13389-020-00233-8, https://link.springer.com/10.1007/s13389-020-00233-8
31. Slpsk, P., Vairam, P.K., Rebeiro, C., Kamakoti, V.: Karna: A Gate-Sizing based Security Aware EDA Flow for Improved Power Side-Channel Attack Protection. In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 1–8 (Nov 2019). https://doi.org/10.1109/ICCAD45719.2019.8942173
32. Snyder, W.: Verilator (2023), https://veripool.org/verilator/
33. Talaki, E.B., Savry, O., Bouvier Des Noes, M., Hely, D.: A Memory Hierarchy Protected against Side-Channel Attacks. Cryptography **6**(2), 19 (Jun 2022). https://doi.org/10.3390/cryptography6020019, https://www.mdpi.com/2410-387X/6/2/19
34. Veshchikov, N.: SILK: High level of abstraction leakage simulator for side channel analysis. In: 4th Program Protection and Reverse Engineering Workshop. pp. 1–11. New Orleans, Louisiana, USA (Dec 2014)
35. Waterman, A., Asanović, K., Hauser, J.: The RISC-V Instruction Set Manual: Volume I, version 20240411 (Apr 2024), https://github.com/riscv/riscv-isa-manual/releases/tag/20240411
36. Zeitschner, J., Müller, N., Moradi, A.: PROLEAD_SW: Probing-Based Software Leakage Detection for ARM Binaries. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 391–421 (Jun 2023). https://doi.org/10.46586/tches.v2023.i3.391-421, https://tches.iacr.org/index.php/TCHES/article/view/10968