Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures

Johan Laurent†, Vincent Beroulle†, Christophe Deleuze†, Florian Pebay-Peyroula‡

† Univ. Grenoble Alpes, Grenoble INP*, LCIS
26000 Valence, France
firstname.lastname@lcis.grenoble-inp.fr

‡ CEA-LETI
38000 Grenoble, France
florian.pebay@cea.fr

***Abstract*—To protect against hardware fault attacks, developers can use software countermeasures. They are generally designed to thwart software fault models such as instruction skip or memory corruption. However, these typical models do not take into account the actual implementation of a processor. By analyzing the processor microarchitecture, it is possible to bypass typical software countermeasures. In this paper, we analyze the vulnerability of a secure code from FISSC (Fault Injection and Simulation Secure Collection), by simulating fault injections in a RISC-V Rocket processor RTL description. We highlight the importance of hidden registers in the processor pipeline, which temporarily hold data during code execution. Secret data can be leaked by attacking these hidden registers. Software countermeasures against such attacks are also proposed.**

Keywords—Fault attack, Fault modelling, Microarchitecture analysis, Hidden register, RISC-V, Software countermeasure

I. INTRODUCTION

Nowadays, hardware fault attacks are a common threat for secure digital systems. To protect against that kind of threat, systems have to be hardened, using both hardware and software countermeasures. Most of the time, these countermeasures are designed without considering how the hardware and the software interact with each other. This absence of interaction analysis between hardware and software can induce low countermeasure efficiency (the system is not well protected against practical fault injections) or induce overprotections and extra costs. Software countermeasures are added in the software application to thwart fault injections based on typical software fault models, such as instruction skip, or register corruption [1]. These models, although useful for their simplicity and genericity, operate at a too high level to be representative of the actual fault injection effects in the processor microarchitecture [2]. In particular, they do not consider hidden hardware processor structures or hardware countermeasures. The same is true for hardware countermeasures which are also generic and do not take into account software applications and their countermeasures.

The main contribution of this work is to show how considering hardware and software specificities together can lead to successful attacks, even in a protected system. This is shown by propagating faults into a RISC-V microarchitecture. The paper highlights the role of hidden registers in particular. Finally, it shows how to protect applications against such attacks, using simple software countermeasures.

The attacks that we will consider will target hidden structures of the processor pipeline. Several works have already demonstrated the importance of fault attacks on hidden structures in a quantitative manner [3,4]. In this paper, we focus on understanding how these hidden structures impact security. More specifically, we focus on hidden registers that we define as sets of invisible flip-flops that store data used during code execution. They are invisible from the software point of view and are completely dependent on processor implementation. We do not refer to general-purpose or special-purpose registers accessible from the instruction set (for clarity, we will call these "processor registers"). We do not refer to flip-flops that store control signals either. Hidden registers store data that have a real meaning in a program; for example, we will see that one of them holds the result of the last multiplication executed. Hidden registers are not always cleared after their use, so they can retain data that have been previously processed. Using fault injection, we are able to re-use these data, for example to leak them.

Faults are injected into an application that comes from the Fault Injection and Simulation Secure Collection (FISSC)[5]. This is a collection of codes that include various countermeasures designed to thwart fault attacks. We will inject faults into a processor called Rocket core. It is an implementation of the RISC-V instruction set [6]. Rocket and RISC-V are both open-source.

The paper is organized as follows. Section II describes the system under attack, showing both hardware and software sides. Then, section III shows three different attacks on the system. Section IV discusses the relevance of these attacks, and proposes a general software countermeasure. Finally, Section V concludes and gives some directions for future work.

II. SYSTEM DESCRIPTION

This section introduces the system under attack. First, the software side is presented, with a description of the application and its software countermeasures. Then, some structures of the hardware architecture are analyzed. These structures will be attacked in next section.

A. Software side: the VerifyPIN program

FISSC is a collection of codes that use different kinds of countermeasures [5]. They are designed to evaluate, on concrete examples, the robustness of software countermeasures against fault attacks. Each code comes in several versions that use various levels of countermeasures. We will focus on the VerifyPIN example, which is a very simple code that compares 4-digit PIN values. A pseudo-code is shown in Listing 1.

* Institute of Engineering Univ. Grenoble Alpes

This work was funded thanks to the French national program 'programme d'Investissements d'Avenir, IRT Nanoelec' ANR-10-AIRT-05.

```
VerifyPIN
Authenticated=FALSE;
If (cnt > 0)
    If(Compare()==TRUE)
        cnt=3 ;
        Authenticated=TRUE ;
    Else cnt-- ;
```

```
Compare
diff=FALSE;  status=FALSE;
For(i=0 ; i<4 ; i++)
    If(userPIN[i] != cardPIN[i])
        diff=TRUE;
If(diff==FALSE) status=TRUE;
Else status=FALSE;
Return status;
```

Listing 1: Pseudo-code of the VerifyPIN application

VerifyPIN comes in eight different versions. We will focus only on the most secure, with the following countermeasures:

- Hardened Booleans: instead of using 0 and 1 for Booleans, values are 0x55 and 0xAA.
- 4-times loop: the loop should always be executed four times. At the end of the loop, the function *countermeasure* is called if the iterator (*i*) is not 4.
- Inlined call: the call to *Compare* is inlined in the rest of the code. This protects against attacks that skip the function call or that target context switch.
- Double test: tests on Booleans are duplicated. If the second test does not produce the same result as the first one, the *countermeasure* function is called.
- Step Counter: Very regularly, a variable named StepCounter is incremented, and is immediately compared to the value it should have at that point in the program. This protects against attacks targeting control flow integrity.

The code is compiled with GCC, with the –Og flag. O1 optimizations are too aggressive in that they remove some countermeasures. On the other hand, the -O0 flag (no optimization at all), gives a code flooded with memory accesses, due to extreme register spilling. The –Og compilation flag seemed like a good middle ground: it does small optimizations, but keeps the structure of the code, so it does not remove the countermeasures.

B. *Hardware side: the Rocket processor and its structures*

In this section, we describe two structures of the processor, as well as the behaviors we can expect from them if faulted. The processor is LowRISC v0.2. It implements a 64-bit Rocket core that has a 5-stage pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory operations (MEM), and Write-Back (WB). We will consider that the processor also includes hardware countermeasures, so that attacks on the register-file, memories (both instruction and data) and control signals driven by instructions are not possible. We will thus focus the attacks on hidden registers of the processor. These structures are not necessarily hardened since it is expensive and implies performance overheads; and other structures are protected in priority, like the register-file or memories [7]. Aside from that, these structures can perhaps be protected in software, which is preferable since it is cheaper and more flexible. In the following, we describe two structures of the processor implementation.

1) Multiplier

To compute multiplications, the processor does not use the ALU, but a separate unit. When a multiplication is required, the processor sends input values to the unit, and tells it in which processor register the result should be stored. The multiplication is controlled with an FSM. After it is finished, the result is written into the destination processor register.

The result of the multiplication and its destination are kept in hidden registers of the multiplication unit, even after the results have already been written. At any time, if a fault impacts the FSM and sets it to its final state, then the unit will again write the same result in the register-file, as if another multiplication was completed. The state of the FSM is stored in a 3-bit signal. Two of these bits have to be impacted.

2) Forwarding

Most modern processors are pipelined: instructions take several cycles to execute. But very often, an instruction needs the result of a previous instruction which is not completed yet. To avoid wasting cycles, a common optimization is used: forwarding. It consists in feeding the output of a pipeline stage back to a previous stage, thus bypassing the register-file. Figure 1 shows how this forwarding is implemented in the Rocket processor. Each of the two arguments of the ALU has the forwarding structure shown in Figure 1 (in other words, this structure is used twice in the processor). FWD is a one-bit register that is set to 1 if forwarding is needed. The use hidden registers MSB(62 bits) and LSB (2 bits) is described below.

When there is no forwarding, the argument of the ALU is read from the register-file (Reg-file). In the Decode stage, the value is separated into its most and least significant bits. These two parts are stored into hidden registers MSB and LSB, respectively. They are then reassembled in the Execute stage and fed to the ALU.

When there is a forwarding situation, the FWD register is set to 1. LSB is set to a value that selects which value should be sent to the ALU. If LSB is “00”, it will forward the value 0. If “01” it will forward the value in the Memory stage. If “10”, it will forward the value in the Write-Back stage. If “11”, it will forward a value that comes from the data cache (D-cache). MSB is simply not used in this case. That means that it is not updated. It will keep its value.

Different means of attack can be thought of in this structure. We will only consider the following two observations that will be used in next section.

- The last value read from data cache remains available for forwarding. At any time, if a fault can activate forwarding with LSB at “11”, the last data cache value can be re-used.
- By de-activating the forwarding in the Execute stage, hidden registers LSB and MSB are concatenated to form the argument. But if a forwarding was detected during the Decode stage, LSB is equal to the stage to be forwarded, and MSB holds its previous value (it has not been updated). So what is important here is that we can re-use the most significant bits of a previous value that was held by MSB.

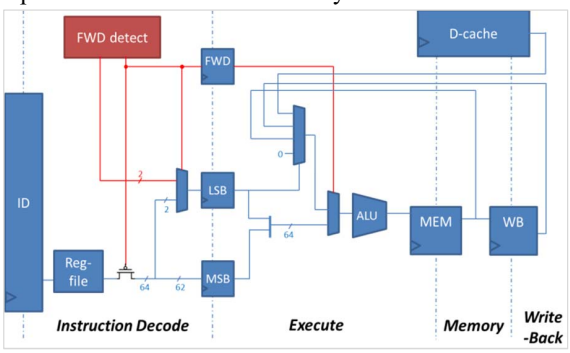

Figure 1: Microarchitecture of the forwarding structure in our processor.

This section showed the role of several hidden registers in the structures of the Rocket processor. As described, an attacker could inject faults into these structures to induce interesting behaviors in a program. We will now see, in next section, how these behaviors can weaken the security of an actual application.

III. Attacks on VerifyPIN and Countermeasures

In this section, we show how to attack the system described in previous section (Rocket processor with the hardened VerifyPIN code). Faults are injected during the RTL simulation of the code execution, by forcing signals with a tcl command that sets a signal to the desired value, until this signal is driven by another source. Each time, a countermeasure is proposed.

A. Attack on the multiplier

For the first attack, we assume that the last multiplication produced the result 0xAA. This value remains at the output of the multiplication unit. By using the fault described in previous section, an attacker can then write this value in a processor register, pretty much at any time. In VerifyPIN, 0xAA represents a Boolean true, according to the Hardened Boolean countermeasure. Attackers can thus use this fault to force a processor register to true. In particular, they could force the authentication by setting the *status* variable to true.

The interesting aspect of this attack is that it is difficult to avoid with standard countermeasures, because the multiplication unit can ask to write in the register-file at any time. To protect against this kind of fault with a software countermeasure, the most effective way would be to empty the hidden register that is problematic. To do that, a dummy multiplication should be computed at the beginning of the code. This way, the hidden register can be set to a neutral value that cannot harm the program.

B. Attack on forwarding: MSB hidden register

In this second attack, we will consider the loop that compares the PIN digits. The assembly code of this part of the program is shown in Listing 2. There are four digits, so the loop should be executed four times. If it is not executed four times, then the function *countermeasure* is called after the loop (not represented in Listing 2). By analyzing Listing 2, we can deduce that from line 7 to line 15, hidden register MSB of the second argument holds the 62 most significant bits of a secret digit. The attack shown in this section consists in leaking this information, so that we know if a secret digit is in the range 0-3 or 4-9. This attack can divide the search space of a digit by 2, but cannot force authentication alone. The attack relies on the behavior of the *countermeasure* function. We will deduce information from the fact that *countermeasure* is triggered or not. We will consider that *countermeasure* blocks the device for a certain amount of time. In other words, we give the attacker the power to distinguish the triggering of *countermeasure*, and to still be able to use the device.

In a fault-free execution, at the end of the loop, there is a comparison between the loop iterator, *i*, and the value 3. If 3 is greater than or equal to *i* then a new loop iteration should start. We know that the loop should be executed four times, so under normal circumstances, the condition should be true in the first three rounds and false in the last. A deviation from this behavior triggers *countermeasure*. With a fault attack, it is possible to alter the condition and make it dependent on secret information. Looking at the code, we can see that for the comparison instruction (line 15), the value of *i* is forwarded; indeed, its value is computed only two instructions before (line 13). Using a fault, we can deactivate the forwarding. So instead of comparing the value of *i*, we will compare the hidden register MSB, concatenated with "10" (value of LSB when the WB stage is forwarded). The comparison then becomes:

Is $3 \geqslant (\text{MSB} \mid \text{“10”})$?

This condition is true if MSB is 0; false otherwise. So, during the first three rounds, if MSB is not 0, the loop is left prematurely, triggering *countermeasure*. Similarly, during the last round, if MSB is 0 the loop is left too late, also triggering *countermeasure*. By observing the triggering of *countermeasure*, we can thus deduce information about what is in MSB, i.e. the most significant bits of a secret digit (as a result, we know if the secret digit is in the range 0-3 or 4-9).

To protect against this attack, a simple way would be to swap the comparison of digits. Instead of doing:

If (userPin[i] != cardPin[i])

It is possible to prevent the attack by doing:

If (**card**Pin[i] != **user**Pin[i])

Indeed, doing that, the user PIN is loaded after the card PIN. So, from line 7 to 15, MSB would hold the user digit (which is not secret) instead of the secret digit. The attack would thus leak meaningless information.

```
1   loop:  addi   a4, gp, -2008
2          add    a4, s0, a4      Load userPIN[i] in register a3
3          lbu    a3, 0(a4)
4          addi   a4, gp, -2016
5          add    a5, s0, a4      Load cardPIN[i] in register a5
6          lbu    a5, 0(a5)
7          beq    a3, a5, step    if (a3==a5), skip next instruction
8          addi   s2, zero, 0xAA  Digits not equal : set diff to TRUE
9   step:  addiw  s1, s1, 1
10         addiw  a5, s0, 4       StepCounter check
11         beq    s1, a5, inc
12         jal    ra, 564
13  inc:   addiw  s0, s0, 1       i++
14         addi   a5, zero, 3
15         bge    a5, s0, loop   if(3 ≥ i), do another loop iteration
```

Listing 2: Part of the code that compares digits. On the left is the assembly code, and on the right the corresponding behavior.

C. Attack on forwarding: value read from memory

Before describing the last attack, we have to consider how the *VerifyPIN* function is used in the main function. We have seen that inlining a function can be a way to thwart attacks that aim at completely bypassing the function, or attacks that target context switch. Inlining is also a common optimization used by compilers to achieve better performances. So there are security and optimization reasons to use inlining. We will then consider that *VerifyPIN* is inlined in the main function. The third attack uses the fact that the last value loaded from memory remains in a hidden register, and this value can be forwarded at any time. Because secret digits are read after user digits, the former ones remain for a relatively long period of time available in a hidden register, as shown in Figure 2. Moreover, because digits are stored in bytes, the whole PIN is in the hidden register, and not only each digit separately (this is because the memory reads whole words). When *VerifyPIN*

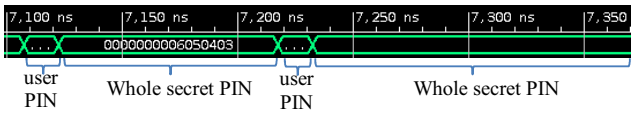

Figure 2: Chronogram showing that the whole secret PIN (3456) remains visible for long periods of time. It can be forwarded (with LSB at "11").

is exited, the hidden register still holds the whole secret PIN. And this information can be forwarded at any time by injecting a fault that activates the forwarding with LSB at "11". So the secret value could be used in a wide variety of things outside *VerifyPIN*.

To protect against this kind of vulnerability, the same countermeasure as in previous section can be formulated. By swapping the arguments in the digit comparison, the PIN that would be leaked would be the user PIN, which is not secret. And using integers instead of bytes would limit the amount of information that can be recovered: in this case, only the last digit would leak to outside of the program.

Another consideration is that if *VerifyPIN* is not inlined, then there would be context switches at the beginning and the end of the function. These context switches use memory operations. So, in this case, context restauration can be seen as a countermeasure since it overwrites the hidden register containing secret information. Inlining can be dangerous regarding hidden registers, because it reduces independence between functions (context switch can be a way to separate functions). Interestingly, inlining can create a similar issue in memory. In [8], D'Silva et al. highlight the fact that inlining can eliminate the boundaries between stack frames.

IV. Discussion

A. Relevance of these attacks

One could argue that the attacks shown are too specific: they cannot succeed under any circumstances. The code that is used here is a small application that was already hardened against fault attacks. It is thus no surprise that it is difficult to find very effective attack paths. The fact that we can find some vulnerabilities in such a code shows that hidden registers, and more globally the processor microarchitecture, should be taken into account when designing software countermeasures. In the end, this paper is meant to raise awareness on the kind of security issues hidden registers can bring. The circumstances in which the attacks arise should be watched closely, and care should be taken to avoid them.

These attacks also show that hidden registers can reduce boundaries between programs. For example, attack A is linked to information from before the VerifyPIN code, and attack C is linked to what happens after the code is executed. Information from one program can leak into another program through these hidden registers.

B. Countermeasures against hidden registers

To prevent these attacks, several countermeasures have been proposed in previous sections. Sometimes, slight alterations of the code can prevent dangerous behavior: for example, swapping the arguments userPin and cardPin. However, these countermeasures are really specific to the code that was analyzed. To protect in a more general way, the best approach would probably be to empty all hidden registers – or, more precisely, fill them with neutral data.

To do that, we would have to add dummy instructions: a dummy multiplication to empty the output of the multiplier unit and a dummy load to empty the output of the data cache. Emptying MSB is slightly trickier: a dummy value from the register-file should be read (for both arguments), while making sure to not trigger forwarding (otherwise, MSB would not be updated). All these dummy instructions should be executed before and after each critical section of the code. In this way, critical sections of the code are decoupled from data that are outside of the critical section. Interestingly, a similar countermeasure is widely used in general software security: memory overwriting. It consists in executing dummy writes so that secret information in memory is overwritten. With hidden registers, the threat is similar: persistent secret data.

V. Conclusion and Perspectives

In this paper, we have looked at different attacks that make use of hidden registers in a processor. These structures can prove to be vulnerabilities even for a hardened application such as the PIN verification we used. As a result, they should be taken into account when designing software countermeasures. Hidden registers throughout the pipeline reduce the independence between successive instructions: information from previous instructions can be re-used by an attacker. Using fault injection, he/she can make an instruction have an impact much later in a program. To protect against this behavior, the best solution would be to add code sequences that clear hidden registers.

A first perspective of this work would be to analyze a more complex example code. Such a code could reveal more interesting attacks. Another perspective is to build a tool that could automatically detect potential vulnerabilities in a code, from a low-level fault model extracted from a processor microarchitecture description.

References

[1] Joint Interpretation Library, "Application of Attack Potential to Smartcards." Jan-2013.

[2] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, "On the Importance of Analysing Microarchitecture for Accurate Software Fault Models," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, 2018, pp. 561–564.

[3] M. Rebaudengo, M. S. Reorda, and M. Violante, "Analysis of SEU effects in a pipelined processor," in *Eighth IEEE International On-Line Testing Workshop (IOLTW)*, 2002, pp. 112–116.

[4] D. Gil, J. Gracia, J. C. Baraza, and P. J. Gil, "Analysis of the influence of processor hidden registers on the accuracy of fault injection techniques," in *Ninth IEEE International High-Level Design Validation and Test Workshop (IEEE Cat. No.04EX940)*, 2004, pp. 173–178.

[5] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "FISSC: A Fault Injection and Simulation Secure Collection," 2016, pp. 3–11.

[6] "The RISC-V Instruction Set Manual," *RISC-V Foundation*. [Online]. Available: https://riscv.org/specifications/. [Accessed: 28-Mar-2018].

[7] A. Benso, S. D. Carlo, G. D. Natale, and P. Prinetto, "A watchdog processor to detect data and control flow errors," in *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*, 2003, pp. 144–148.

[8] V. D'Silva, M. Payer, and D. Song, "The Correctness-Security Gap in Compiler Optimization," in *Proceedings of the 2015 IEEE Security and Privacy Workshops*, Washington, DC, USA, 2015, pp. 73–87.