Insertion of random delay with context-aware dummy instructions generator in a RISC-V processor

Gaetan Leplus, Olivier Savry, Lilian Bossuet

Insertion of random delay with context-aware dummy instructions generator in a RISC-V processor

Gaëtan Leplus
Univ. Grenoble Alpes
CEA, Leti,
F-38000 Grenoble, France
gaetan.leplus@cea.fr

Olivier Savry
Univ. Grenoble Alpes
CEA, Leti,
F-38000 Grenoble, France
olivier.savry@cea.fr

Lilian Bossuet
Laboratoire Hubert Curien
Jean Monnet University
F-42000 Saint-Etienne, France
lilian.bossuet@univ-st-etienne.fr

***Abstract*—Embedded systems are vulnerable to side channel and fault injection attacks. These two types of attacks can be slightly complicated by using temporal desynchronization methods. In this article we propose a new hardware solution to efficiently insert dummy instructions in run time for a general-purpose processor. The main contribution of this solution is to contextualize these dummy instructions, making them less distinguishable and more variable with a minimal spatial overhead of 2.96% and a 4.27% additional consumption and no code size impact on a CV32E40P RISC V processor. As a result, they bring a significant resistance to resynchronization methods.**

Keywords*—*Side channel attacks, Fault injection attack, countermeasures, random delays

I. INTRODUCTION

Side-channel attacks such as differential power analysis (DPA) or fault injection represent a threat to embedded implementations of cryptographic algorithms. Most side channel and fault injection attacks require the adversary to know when the target operations occur during execution. This allows the synchronization of multiple traces at the critical event, as in DPA [1] or template attacks [2], or the introduction of a perturbation into the computations at the appropriate time, as in fault injection attacks [3]. The synchronization can be disrupted by random delays in the execution, increasing the complexity of the attack. Therefore a simple but effective physical attack mitigation strategy is to incorporate random delays into the execution of a cryptographic algorithm. It is a sort of concealment countermeasure that adds noise to side channel leaks (in the time, amplitude, or frequency domain) without deleting any information from the signal itself.

Among the various temporal randomization techniques proposed in the literature [4] there is a general distinction between software techniques based on random delay interrupts (RDI) [5] and hardware techniques based on increasing clock jitter. In general, the more countermeasures modify the operating parameters of the hardware e.g. clock jitter the more the solutions are focused on signal processing. In this context, it is worth noting that many evaluations of countermeasure effectiveness, preprocess the leakage traces by integrating them [6]. Somewhat influenced by this evaluation technique, researches are being conducted to increase the variability of dummy cycle insertions as much as possible in order to improve the statistical distribution of the random sample of delays, resulting in the noisiest dummy cycle insertions [5]. Other pattern recognition-based evaluations, on the other hand, can eliminate instruction additions [7].

In [7], the authors propose requirements for an efficient insertion of dummy instructions, they should use :

- delays with no regular pattern
- insertions that are not predictable
- delays that look like the surrounding instruction

Regular patterns are difficult to prevent for software methods since these instructions frequently require prologues. Furthermore, in many applications, we cannot afford to increase code size. Only hardware runtime insertion of dummy instructions allow to avoid this increase. Nevertheless, hardware solutions from the state-of-the-art, such as [8], [9], answer the first two points by randomly inserting instructions without pattern, but they raise the issue of distinguishing between dummy and genuine instructions.

The proposed hardware solution in this article not only addresses the issue of context-consistent instruction, but it also addresses how to make our dummy instructions less differentiable and more

diversified than other solution with a smaller hardware footprint and having no impact on the code size.

II. THE PROPOSED COUNTERMEASURE

The proposed countermeasure is a hardware system able to insert dummy instructions at random intervals while the program is running. The goal is to materialize the concept of a dummy cycle, which is used at compile time in the vast majority of proposals. To facilitate explanations of the realization, we place ourselves in the RISC-V architecture with a CV32E40P in order processor with four stages. The choice of instruction set architecture (ISA) and processor architecture does not affect the relevance of the solution.

The architecture of the proposed countermeasure, presented on Figure 1, is divided into two parts: the generation of the instruction with the register "dummy opcode" and the block “Generate random instr”. Then the injection of the instruction into the processor pipeline with a multiplexer is controlled by a programmable frequency divider “Variable div clock” and a random bit “RNG”. There is also the propagation of a “flag dummy” signal to warn the following stages if a dummy instruction is in progress.

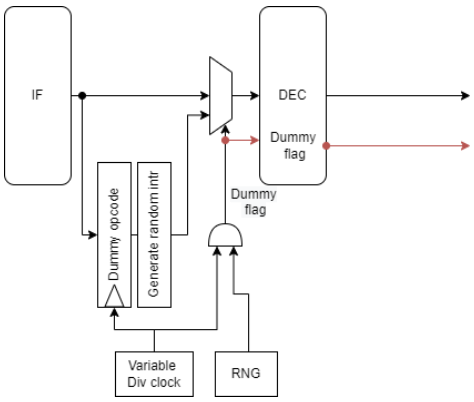

Figure 1: Architecture dummy instruction insertion

A. Instruction generation

Dummy instructions generation should conduce to obtain instructions similar as much as possible to valid instructions, but it is challenging. The most common solution is to create the dummy instructions from predefined arithmetic opcodes. This solution generates those using already processed instructions, allowing to take advantage of almost the entire ISA as dummy instructions. Only the elements necessary for determining the type of instruction should be saved.

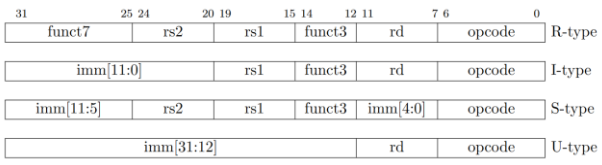

Figure 2: RISCV instruction types

Looking at the various types of instructions in Figure 2, we can see that only 17 bits of field opcode, funct3, and funct7 are useful during instruction decoding. It is important to pay attention to the instructions requiring a particular level of privilege. In RISC-V architecture these instructions are grouped in the SYSTEM opcode and are ignored in our solution to avoid any privilege inconsistencies.

Thus, if only these 17 bits are saved and the remaining bits can be chosen at random, the decoding should always be valid. But generating 15 random bits requires a costly RNG implementation. It is, therefore, preferable to choose these 15 bits in an internal state of the processor (from the control/status register (CSR), the registers, the program counter (PC), etc.). Its selection has few constraints; it just needs to change quite frequently among a large set of values. It has also few security problems because the manipulated data is used for register or immediate selection; it allows different inputs for the instructions at each run, and adds variability by randomizing the Hamming weight of the dummy instruction and the Hamming distance with the next and previous genuine instructions.

B. Instruction insertion

The insertion of the instructions must not be predictable. It is necessary to add randomness in the decision to insert dummy instructions. In our solution, a variable clock divider provides a clock signal at the frequency $F * 2$ when F is the targeted frequency of insertion of dummy instructions. F Can be defined by a configuration register, for example a CSR. In our implementation, for practical reason we use a 32-bit register, outside the processor core, modifiable by JTAG. The insertion of an instruction is conditioned by a multiplexor controlled by the output of a AND gate with the output of a random number generator and the output of the clock divider as inputs. There is a 50/50 chance that the output signal of the clock divider crosses the AND gate, resulting in an average frequency of dummy instruction insertion equal to F. At each period of the clock divider signal, the system recovers one instruction and stores it in the “dummy opcode” register. The variability has increased again because stored opcodes are used only once.

The insertion of the dummy instructions is done at the DECOD stage. At this moment the IF stage is frozen. The execution is resumed in the next cycle in a transparent way for the core. Thus, the normal flow of execution is not disturbed and interruptions are always functional.

With a dummy instruction and the generated signals during its running, we have to make sure that it does not disrupt the normal flow of execution. The “dummy flag” signal informs the processor that it is executing a dummy instruction.

III. PROCESSOR ARCHITECTURE MODIFICATION

It is difficult to define all of the changes because they are heavily dependent on processor architecture and implementation. However, some general considerations can be raised. First, when the processor runs a dummy instruction, no writing to memory or legitimate registers is permitted for obvious memory integrity reasons. In addition, it is necessary to avoid that the processor modifies the control registers, flags, or the PC; for example, jump and branch instructions. It is also required to prevent results from being bypassed. As proposed below, it might be interesting to change the architecture so that these dummy cycles are more similar to legitimate instructions.

A. Writing to registers and memory

Writing to the register bank is a distinguishing factor that can be used to detect dummy instructions. To avoid this, we only write to registers that are not used by the normal flow of execution. There is no need to change anything for the reads in register because only the writes influence on the genuine control flow. Because all of the registers are likely to be used, it is necessary to plan for two additional destination registers (shadow register). In the CV32E40P processor at each cycle, two registers can be written at the same time, one for the result of the ALU and another for memory accesses. However, writing always in the same registers is identifiable. Then, it is possible to implement a dynamic register bank, as shown in [10], to use the same physical registers for dummy and legitimate instructions.

The dynamic register bank, in Figure 3, is made possible by separating the register indexes targeted by the ISA from the physical registers. The index refers now to a lookup table pointing to a physical register location. A validity table of the physical registers must be kept in order to determine which register is utilized.

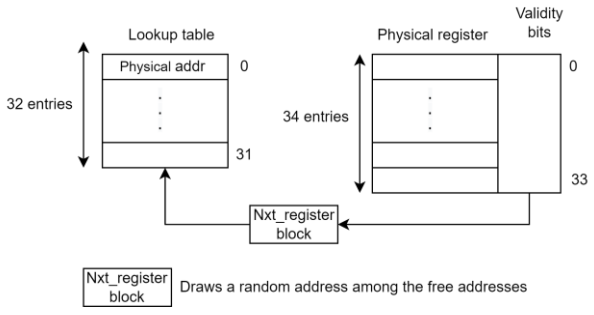

Figure 3 : Dynamic register bank

The validity table consists in adding a validity bit to each register in the bank. When a physical register is written, it is considered valid. There are two ways to invalidate a register, the first is when a function return. Indeed when a "RET" instruction is executed a part of the registers is saved and the other one is not. These unsaved registers can be considered as invalid. The other way to disable a physical register is to maintain the dynamic nature of the register bank. When writing to the register bank, the physical register associated with the ISA register to be written is first invalidated, and then a new physical register is chosen from the free registers. Thus at each writing in an ISA register, this one is written in a different physical register. Figure 4 illustrates a writing in an ISA register already assigned to a physical register.

The "Nxt_register block" draws a register index at random from the invalid registers. The complex logic of selection is disclosed in [10] with an implementation closed to ours. In our case, we have up to two writes in the registers per cycle, so we have to draw randomly two different registers. This drawing was created with the assistance of a second "Nxt_register block" with different input parameters.

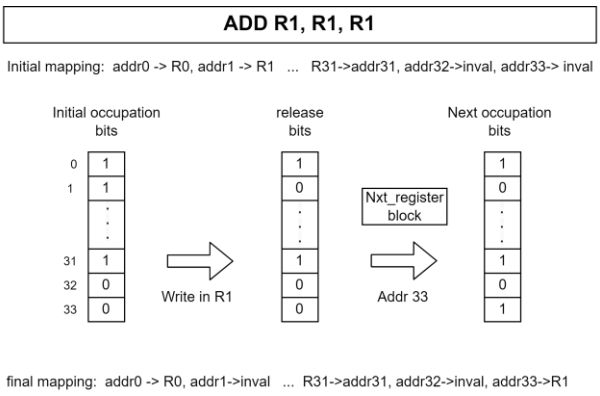

Figure 4: Register validity update

When dummy instructions write to registers, the written register should not be validated. Moreover, it is not required to perform the aforementioned invalidation step. As a result, with dynamic register, the only discernible difference, from the attacker's perspective, for dummy register manipulation is in the management of the validity bits.

The management of Load/Store instructions is a concern because of the difficulty of determining free memory spaces. We solve this issue by assigning a unique address to memory reading and writing.

B. Jumping and branching

Because branches and jumps are common instructions, we must find a way to execute them while remaining as close to the original behavior as possible. We use the same operation for jumps, but instead of jumping to the address indicated in the operand or register, we go to the next instruction.

The same approach cannot be used for branches; in fact, if the branch is taken, the processor must cancel the instructions that were executed prior to the result of the branch. As a result, instructions would be re-executed, posing a problem not only of detection but

also of leakage via side channels. As a result, we were forced not to consider the branches as dummy instructions.

IV. Discussion

Like most hardware solutions, the proposed solution of dummy instruction insertion, deals with both non-use of regular patterns and unpredictable insertion. However, it is the first solution which makes dummy instructions consistent with the current program with the reuse of already executed legitimate instructions and then able to reach the requirements for an efficient insertion of dummy instructions as defined by [7] and without increasing the code size.

In comparison to other hardware methods, our solution allows us to insert all logical and arithmetic operations, jumps, branches, and memory accesses, whereas other solutions insert only a few arithmetic instructions which limits their use to cryptographic primitives. In addition to time desynchronization, our solution includes dynamic register implementation, which complicates side-channel and fault attacks on the register bank. We implemented our architecture in RTL based on RISCY CV32E40P processor and synthesized it by RTL synthesis based on GF22FDX (GlbalFoundries 22nm FD-SOI) Standard Cells RVT process library. The results showed that it needed 15395 GEs with shadow register or 17540 GE with dynamic register to implement our secure processor core. There was about respectively 2.96% and 14.83% additional area that was required and is also an increase in total power consumption of 4.27% and 10.19% over the original RISCY core which has an area of 14938 GE. As a comparison the overhead is 6.33% for softRIJJD and 8% for ERIST for an implementation on an ARM7 architecture of initial size 30081GE synthesized it through the Synopsys Design Compiler based on a UMC 0.18 µm standard cell process library[8].

As far as the performance overhead is concerned, it depends on the level of security that needs to be reached and the overhead of execution time that is willing to accept. The most common ratio in other solutions is between 15 and 25% [8], [9].

Another issue that has received little attention and evaluation is distinguishing between dummy and genuine instructions in hardware. The classical methods of resynchronization such as cross-correlation or hidden Markov models do not seem to be able to differentiate them [8]. However, neural networks could be trained to suppress these instructions by learning the induced difference, in the same way of what is done for the clock jitter based countermeasures [11]. Although we propose improvements to make it more difficult, it remains to be assessed whether this difference is significant enough for the various types of instructions to be used to resynchronize the traces.

V. Conclusion

In this paper, we presented a hardware solution to insert dummy instructions at run time that is optimized for general-purpose processors and has a low overhead. It consists in inserting a random delay based on dummy random instructions depending on the execution context. To perform it, the proposed solution involves the insertion of randomized instructions that have already been used without prologue or preamble, places us in an optimal framework that software solutions cannot reach. Unlike other hardware solutions, we propose instructions that are consistent with the execution context, and our instructions are highly variables.

The remaining work is to assess the impact on leakage of greater variability in the dummy instructions, but more importantly, assessing the feasibility of resynchronizing the traces based on the distinction between genuine and dummy instructions.

Acknowledgment

This work was supported by the French National Research Agency in the framework of the "Investissements d'avenir" program (IRT Nanoelec, ANR-10-AIRT-05).

References

[1] P. Kocher, J. Jaffe, et B. Jun, « Differential Power Analysis », in *Advances in Cryptology — CRYPTO' 99*, Berlin, Heidelberg, 1999, p. 388-397. doi: 10.1007/3-540-48405-1_25.

[2] S. Chari, J. R. Rao, et P. Rohatgi, « Template Attacks », *Lect. Notes Comput. Sci. Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinforma.*, vol. 2523, p. 13-28, 2003, doi: 10.1007/3-540-36400-5_3.

[3] N. F. Ghalaty, B. Yuce, M. Taha, et P. Schaumont, « Differential Fault Intensity Analysis », in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, sept. 2014, p. 49-58. doi: 10.1109/FDTC.2014.15.

[4] C. Clavier, J.-S. Coron, et N. Dabbous, « Differential Power Analysis in the Presence of Hardware Countermeasures », in *Cryptographic Hardware and Embedded Systems — CHES 2000*, Berlin, Heidelberg, 2000, p. 252-263. doi: 10.1007/3-540-44499-8_20.

[5] J.-S. Coron et I. Kizhvatov, « An Efficient Method for Random Delay Generation in Embedded Software », in *Cryptographic Hardware and Embedded Systems - CHES 2009*, Berlin, Heidelberg, 2009, p. 156-170. doi: 10.1007/978-3-642-04138-9_12.

[6] S. Mangard, « Hardware Countermeasures against DPA – A Statistical Analysis of Their Effectiveness », in *Topics in Cryptology – CT-RSA 2004*, Berlin, Heidelberg, 2004, p. 222-235. doi: 10.1007/978-3-540-24660-2_18.

[7] F. Durvaux, M. Renauld, F.-X. Standaert, L. van Oldeneel tot Oldenzeel, et N. Veyrat-Charvillon, « Efficient Removal of Random Delays from Embedded Software Implementations Using Hidden Markov Models », in *Smart Card Research and Advanced Applications*, Berlin, Heidelberg, 2013, p. 123-140. doi: 10.1007/978-3-642-37288-9_9.

[8] Z. He, T. Ao, M. Wan, K. Dai, et X. Zou, « ERIST: An Efficient Randomized Instruction Insertion Technique to Counter Side-Channel Attacks », *IAENG Int. J. Comput. Sci.*, vol. 43, p. 65-71.

[9] J. A. Ambrose, R. G. Ragel, et S. Parameswaran, « Randomized Instruction Injection to Counter Power Analysis Attacks », *ACM Trans. Embed. Comput. Syst.*, vol. 11, nº 3, p. 69:1–69:28, sept. 2012, doi: 10.1145/2345770.2345782.

[10] D. May, H. Muller, et N. Smart, « Non-deterministic Processors », juill. 2001, vol. 2119, p. 115-129. doi: 10.1007/3-540-47719-5_11.

[11] E. Cagli, « Feature Extraction for Side-Channel Attacks », 2018.