SecDec : Secure Decode Stage thanks to masking of instructions with the generated signals

Gaetan Leplus, Olivier Savry, Lilian Bossuet

SecDec: Secure Decode Stage with Instructions Masking

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 – Physical attacks are becoming a major security issue in IOT applications. One of the main vectors of attacks on processors is the corruption of the execution flow. Fault injections allow the modification of instructions, in particular jumps and branches. The proposed approach involves making a RISC-V processor's instruction path more resistant by introducing dependencies between succeeding instructions. The signals extracted from the instruction decoding stage is used to unmask the following instruction. Whereas all instructions have been previously masked during compilation with the expected mask. We show that this solution has a very low hardware overhead of 3.25% and power consumption of 4.33%. But also overhead software of 1.8% in code size and 1.11% in execution time. An instruction corruption or a jump will be detected on average in fewer than 2 cycles after the fault while making disassembling from side-channel leakages becomes more difficult.

Keywords – Fault injection attack, countermeasures, masking, fault detection

I. INTRODUCTION

Embedded applications require hardware that is both lightweight and secure in order to preserve the confidentiality and integrity of the executed code. Physical and observation attacks are growing against IoT processors and the proposed countermeasures often have a significant overhead. Solutions to protect each of the stages in the most efficient way must be found by taking advantage of their specificities. The CPU front end, in particular, is sensitive to fault injection attacks, as it is the center of instruction management. Indeed, fault injections are particularly effective on instructions, especially faults generating instruction jumps [1], [2] which allow overriding many algorithmic securities by changing the computation result or the execution flow. The latter focuses in particular on the modification of branching or instruction modification to be able to make jumps in the code. But these are not the only threats that are identified. The intellectual property of source code can also be challenged with, for example, side channel disassembly [3].

To ensure the security and safety of the instruction's execution against previous threats, four properties must be respected:
—The current instruction is well preceded by the previous instruction in the execution flow. This means avoiding jumps and instruction modifications.
—The instruction decoding stage (DECOD stage) performs the instructions decoding without errors.
—The processor must execute the right branch.
—The value of each bit of the instructions must not be directly accessible by side channel analysis.

In the state of the art, those four properties are never addressed efficiently at the same time. The most common solutions to protect the execution flow are control flow integrity (CFI) units [4], [5] which can guarantee the integrity of instructions up to the first stages of the pipeline [6] and can avoid leakage through side-channels in the memory hierarchy if encryption methods are implemented [7]. However, these countermeasures do not protect instruction decoding and thus branching. This type of countermeasure has a significant overhead, because it addresses other attack vectors on the execution flow than physical attacks. In reliable processors, such as Klessydra [8], we find spatial duplication of the DECOD stage to secure decoding. In addition, temporal duplication makes it possible to avoid instruction skipping [9]. The main problem of duplication-based solutions is the hardware and time overhead. Finally, to counter side-channel attacks, masking solutions are the most used, it consists of applying a random variable with a function, in the most classical case it is the function *xor*. However, the implementation costs are significant, especially with the duplication of the instruction size to store the mask.

Our contribution is to give two close solutions, but with different implementation implications. However, these two solutions do ensure the security of the instruction path against perturbation and observation attacks. The first solution has a negligible hardware and instruction memory size overhead, but induces a dependency of the compiled code with microarchitectural constraints. The second solution is free of these constraints, but involves a decoding redundancy. These solutions lead to modifications in the compiler backend, but they are simple to implement and are adaptable to different types of compilation.

This paper is structured as follows: Section II presents our security model that we use in this work and the attacker model that we seek to defend against. Section III presents the detection method of our solutions as well as the generation of masks. Section IV presents the securing of DECOD stage and reasons for proposing two countermeasures that are quite similar, but with different implementation constraints. Section V presents the problems generated during compilation and how they are solved for the proper functioning of our solution. Section VI details the implementation of one of the solutions in a RISC-V core with its overhead. Finally, Section VII evaluates the detection capability of our countermeasure and discusses the resistance against disassembly by side channels.

II. ATTACKER MODEL

This work concerns processors mainly intended for the IoT that are easily accessible to attackers to perform physical attack. This processor can be in a complex platform [10] with an arbitrary memory hierarchy without affecting our security model.

These physical attacks involve direct access to the device to corrupt instructions, dump and manipulate external memory or force signals. In addition, side-channel attacks are also considered. This attacker only has the instruction path as an attack surface, i.e. the instructions in the entire memory hierarchy until the end of their decoding in the processor core. Thus, probing and fault injections on stage other than the fetch instruction and DECOD instruction are not considered. Thus, the entire instruction path is considered sensitive to attacks. The attacker has the whole memory hierarchy, but also the processor pipeline until the instruction decoding to realize his attack.

Our attacker is thus limited to attacks by perturbations (fault injection) and by observations (attacks by side channels). Thus, cold boot attacks [11] are not taken into account as well as software attacks, such as code injection with, for example, the overwriting of the return address on the stack. For this reason, encryption and control flow methods allow a better response to this type of threat. The attacker's objectives in this type of attack are to extract industrial property using side channel disassembly [3], and to bypass security by skipping instructions or corrupting instructions, in other words, to compromise the confidentiality and integrity of the code executed by the device.

III. PRESENTATION OF OPERATION

A. *Proposed countermeasure description*

This paper proposes a countermeasure to the previously presented attack model which consists of using signals generated by the previous instruction to mask the current instruction. It is a non-random Boolean masking, because it depends on the previous instruction. To do this, a mask is generated at cycle N which will be used in the DECOD stage at cycle N+1 to unmask the incoming instruction. We can choose to get signals before the decoder, see Figure 1, it will be the instruction itself, and we call this solution pre-decoder. It is also possible to take the signals after the decoder, see Figure 2, this solution is called post-decoder. Thus the instruction must be masked when it arrives in the DECOD stage. To apply this mask, additional compilation passes must be added in the compiler backend. As a consequence, all

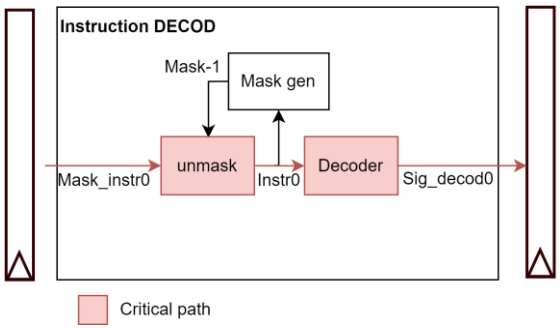


Figure 1: Signals Pre-Decoder

instructions depend on the instruction that precedes it. Thus, if a fault occurs at cycle N the unmasking of the instruction in the decoding stage at cycle N+1 would be different from the decoding without faults. This solution thus allows to check that the present instruction is executed after the instruction which precedes it in the machine code, therefore, one avoids the jumps of instructions and the modifications of instructions. When branching, two paths are possible. Thus, it is common to try during a fault injection campaign to make the processor take the wrong branch to execute unwanted code. The proposed solution makes it possible to differentiate the two branches by assigning them different masks according to the branch taken. In fact, even if the output signals of the DECOD stage are identical, whatever the branch chosen, the outputs of the EXEC stage are different. The final choice of branch is determined when the conditions

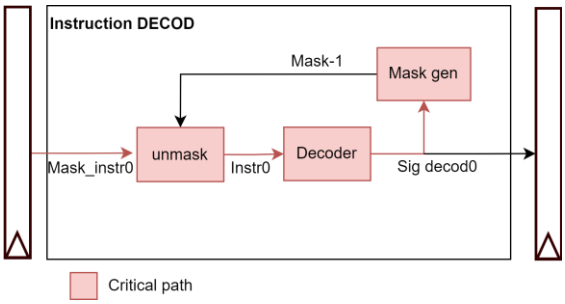


Figure 2: Signals Post-Decoder

of the branch are verified. The choice is therefore made when the result of the branch condition is available, which is normally the case at the end of the EXEC stage. Thus, contrary to the existing CFI which is at least sensitive during the decoding of the instruction, the validity of the branch is ensured by the entire instruction path. The only way to make a branch take the wrong branch is to fault the EXEC stage. This is a case outside of our attacker model, however, duplication or verification solutions of the comparison operations of the EXEC stage to secure the branches [12].

B. *Proposed fault detection method*

The proposed fault detection method is based on the ability of the processor to detect an invalid instruction. Indeed, it is not possible with this method to detect the fault

in itself. The detection is made on the errors of decoding induced by this fault. A fault on the instruction to decode, or on its mask, induces a modification of the decoding. If this decoding is invalid, an "invalid instruction" exception is

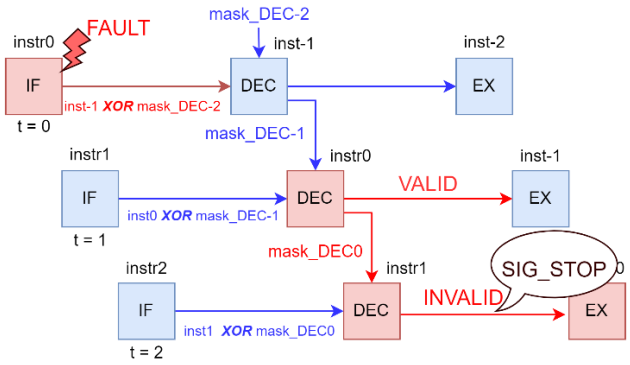


Figure 3: Propagation and detection of a faulted instruction

raised and the error will be detected. However, even if this faulted instruction is valid (i.e. part of the instructions set), the mask generated by this instruction will be different from the instruction that is not faulty. Hence, the more the instruction set is sparse, the more effective this detection. method is.

If a large part of the opcodes is invalid, the detection only takes a few cycles. Figure 3 shows the propagation of a fault through the different stages, fetch instruction (IF), decoded instruction (DEC) and EXEC of a general-purpose processor. Thus, if a fault occurs at time $t = -1$ it generates a false mask which is propagated to the following cycle, but without error detection at the DECOD stage. The same is true of the DECOD stage at cycle 0. However, at cycle 1 the DECOD stage detects an invalid instruction and raises an invalid instruction exception and the corresponding stop signal, so in the example of the Figure 3, the processor takes 2 cycles to detect the fault.

For this work, the proposed countermeasure is implemented in a RISCY RISC-V processor, but the approach is, of course, transposable to another ISA or architecture. It is therefore necessary to see where the errors must be placed to be likely to be identified. In our case, it is the opcode bits, funct3 and funct7. It is, however, to be noted that these bits do not allow the detection of the faults of all the types of instruction. Indeed as one can see in Figure 4, there are several types of instruction, only the type R has 3-field opcode, func3, func7. These are arithmetic and logical instructions. Moreover, the bits of registers (rd, rs1, rs2) because they correspond to the registers or to immediate are always valid and do not allow to detect faults. So, the main sources of fault detection are the opcodes.

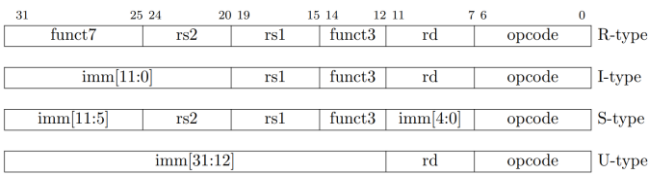
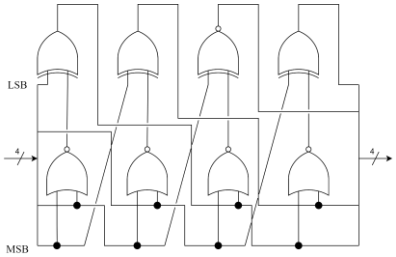


Figure 4: RISC-V instruction type

C. *Mask choice*

In order for this detection to be as fast as possible, i.e. for a fault to result in an invalid instruction as quickly as possible, two complementary approaches can be followed. The first is to make the bits interdependent. If a fault occurs, it must lead to a maximum number of modifications to the bits capable of causing a decoding error. This is more precisely the opcode bits and to a lesser extent func3 and func7. The other approach is the propagation of faults in each new cycle.

For an efficient detection if a modification occurs, the generated mask must be very different from the legitimate mask, but also give a more important place to the opcode bit. To do this, we will generate two masks using the 4-bit sbox from the Piccolo light encryption [13] presented Table , for its lightness of implementation in hardware. The first mask takes blocks of 4 consecutive bits to create dependency, the second mask takes 4 bits at index I, i+8, i+16, i+24 with i $\in$ ⟦0,7⟧. As a consequence, almost all bits become dependent on the opcode bits.



x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S [x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 1: Piccolo Sbox

The propagation of the faulty bits in case of a valid decoding succession is ensured by a permutation of the mask which has no hardware overhead. Indeed, it is intended that the faults propagate on the whole mask if the instructions remain valid after several decoding cycles. For diffusion

P	16	7	20	21	29	12	28	17
	1	15	23	26	5	18	31	10
	2	8	24	14	32	27	3	9
	19	13	30	6	22	11	4	25

Table 2: DES Permutation

properties, the 32-bit permutation of the DES encryption algorithm, presented in Table , is chosen.

Therefore, if a fault occurs before the entry of the DECOD stage, it will cause multiple faults in the generation of the mask thanks to the different sboxes. The combination of a permutation and sbox ensures the diffusion of the faults to all the bits of the instruction in a few cycles.

IV. SECURING THE DECOD

To ensure the integrity of the instruction path, it is also necessary to ensure the integrity of the decoding. It can also be the target of the attacker. Two approaches have been studied to solve this problem.

A. *Generation of the mask with post-decoder signals*

This solution uses the post decoder signals to generate the masks. The signals used are the register index, the immediate, multiplexer selector and enable signals. It is difficult to make generalizations because these signals are specific to each architecture but with these we can ensure that the decoding is correct in the case of RISCY. With the solution of the previous instruction, one protects against faults only up to the beginning of the decoding stage just before the decoder, as shown in Figure 1. If the output signals of the decoder are used, it is assured that the decoding has taken place without errors.

In terms of hardware or instruction memory size, this technique has relatively low overhead. However, due to the fact that the masks are generated from the decoding signals, the critical path of this stage is lengthened and can therefore lead to a decrease in the frequency of the processor. Moreover, the compilation is made more complex, because the decoding signals must be generated in a software way. Thus, a decoder similar to the one present in the DECOD stage of the processor must be added to the compiler. These two decoders must generate the same signals so that decoding is possible. The main disadvantage of this solution is to make the compiled code depend not only on the ISA but also on implementation specificities, in our case the decoding of instructions.

To make the implementation of the countermeasure at the compilation level simpler, the choice was made to use directly the previous instructions and not their decoding signals.

B. *Mask regeneration*

If we use the previous instruction, it is no longer possible to ensure the security of decoding natively. It is therefore necessary to take advantage of the various elements already added. The main one is the recording of the generated mask. It would therefore be interesting to check it at the next cycle to ensure decoding.

The input signals of the EXEC stage can be used as input for an instruction encoder. Indeed, there is no loss of information during decoding, only a change of form. It is hence possible to encode the decoding output signals in the form of the original instruction. Once the instruction has been encoded, the mask can be generated and compared with the one contained in the register, Figure 5.

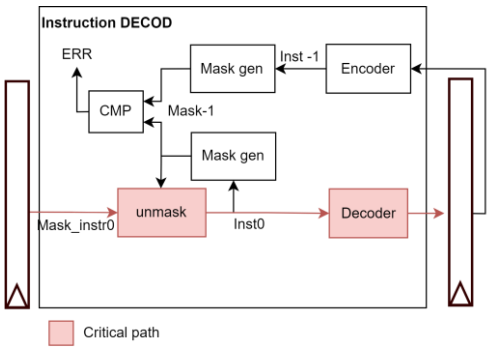


Figure 7: Mask regeneration

This solution can be compared to a duplication of the DECOD stage, but it has advantages compared to a simple duplication. Indeed, the first advantage is to use the signals leaving the register between the DECOD stage and EXEC stage. By using these signals, we also secure this register. Beyond securing the register, it also allows using a simple duplication which is only sensitive to double injection attacks. Multiple fault injections are still considered more difficult to implement [14]. Moreover, compared to simple duplication, a mask generation is used which leads to the diffusion of faults through the whole mask, which further complicates fault detection. The encoding overhead is of the same order as the decoding and is performed in parallel, which does not lead to a lengthening of the critical path compared to duplication.

If we compare with the solution of masks generated from decoding signals, several advantages can be noted. First of all, the detection is no longer limited by the decoding errors and thus allows faster detection of the faults during this one. However, the hardware overhead is more important in this solution with the addition of an encoder and a mask generation, but the critical path is not lengthened.

However, other problems arise if a mask is created from the previous instruction or from the decoding signals. Indeed, the source code of a program is not perfectly linear: jumps are possible at different indexes of the code. Therefore, the previous instruction in the execution thread is not necessarily the previous instruction in the compiled code.

V. COMPILATION PROBLEMS

This masking can only be done at compile time, the only place where the meaning of the assembly code and the sequence of instructions are known. The main difficulty is then to manage the different possible branches of the execution flow. There are two cases, the first is when a basic block has several blocks that can follow each other (Figure 6), and this is the case of branching. To solve this

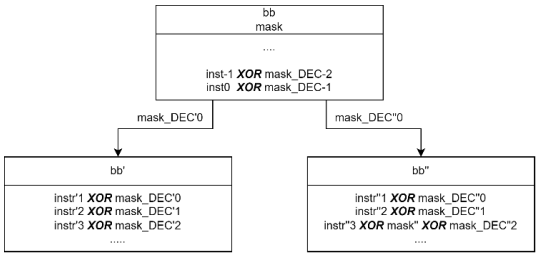


Figure 5: basic block with two successors

problem, we must be able to generate two different masks from the same instruction. The second case arises if several basic blocks point to the same following basic block as shown

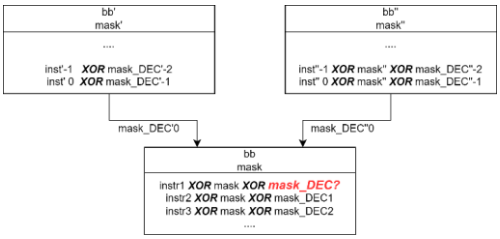


Figure 6: basic block with two predecessors

in Figure 7. Thus an instruction can have several previous instructions and therefore have several possible masks. As the knowledge of the basic block which is at the origin of the preceding instruction in the pipeline of the processor is difficult, one cannot select among the possible masks. In general, only one mask is used per instruction. So, if an instruction has several antecedents, they must all generate the same mask or this instruction must not ask for any mask.

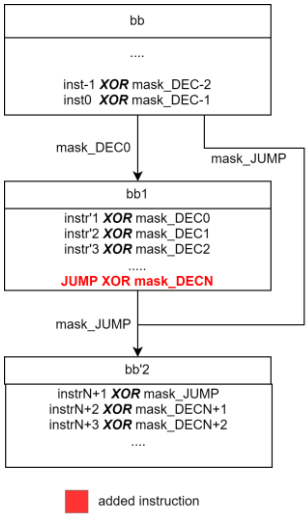


Figure 9: Mask during a branch

A. *Resolving difficulties*

1) Jump

The first instructions that can generate multiple antecedents are jumps, whether direct or indirect, because two jump instructions can arrive at the same address. For this particular case, the mask of the jump instructions must be identical whether the jump is direct or indirect. But it must not depend on the offset or on a register either. Jump instructions with different offsets can arrive at the same instruction.

Another problem is that a jump instruction can also be reached by the standard incrementing of the program. Here the problem is more complex, because any instruction can precede a jump destination. It is therefore necessary that the instruction preceding the masked instruction generates a jump mask. To do this, the solution adopted was to add a jump instruction with the following instruction as a destination before each jump destination. So, a jump destination is only accessible by jumps and the generated mask is always a mask_JUMP, Figure 8.

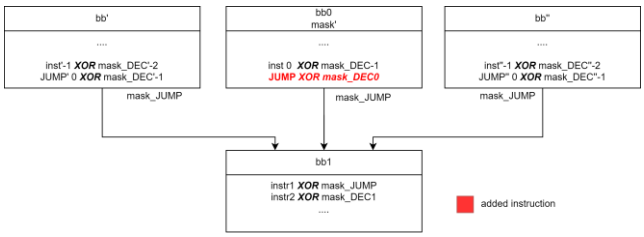


Figure 8: Mask during JUMP

2) Branch

A problem similar to the one raised for jumps remains for branches. Indeed, when the branch is taken, it is comparable to a jump, all the other possible paths to access this instruction must produce the same mask. Adding branches before each branch destination does not make sense, because a branch means a conditional jump, which would be unconditional here. The addition of a jump instruction is therefore recommended. So before each branch destination a jump instruction must be added. Moreover, the mask "mask_DEC0' ", see Figure 9, is replaced by the mask used for jumps, "mask_JUMP", because it is this branch that can be assimilated to a jump.

So, all the instructions causing a jump in the assembly code produce the same mask. Moreover, each jump destination is preceded by a jump, thus allowing having a unique mask, whatever the path to get there. Finally, it is possible to differentiate whether a branch is taken or not by using a different mask in each case.

3) Solving context switching and interrupt problems

When switching context one of the first steps is to save the mask to unmask the next instruction, otherwise it will not be possible to unmask the instruction at the return address. A step of saving the mask is added at the beginning of the context switch and one of restoring the mask at the end of the context switch. This step is analogous to the PC switch. In the case of an interruption, the mask that should have been used in the next cycle if the interruption had not taken place is automatically saved. In the case of RISC-V, this backup is managed automatically at the level of the command and status registers during an interruption.

4) stall solving

A processor for various reasons, data dependency, calculation of the branch to take, etc. can freeze its execution the time that calculations advance in the pipeline to resolve these dependencies. During these cycles, the processor keeps in an internal register during the freeze cycles the mask to use for the next instruction.

B. *Compilation modification*

The changes in the compiler have been made on both GCC and LLVM. Indeed, as previously stated, any instruction that is accessible by a jump or branch must only be accessible by jump instructions. A pass takes all basic blocks with multiple antecedents and checks that each one ends with a jump or branch instruction. If it does not, a jump instruction is added.

Then a pass adds the masks to all the instructions. This pass must be the final pass, because it fixes the order of execution of the program, so the code must not be modified after this pass. This pass is quite simple, because it takes the previous instruction, generates the mask and then applies it to the current instruction. The mask is only a composition of the previous instruction using Sbox and permutation, so it is easy to perform these transformations in software.

The use of constant mask during jumps makes this solution sensitive to instruction jump. Indeed, each JUMP destination instruction has the same mask. Thus, if during an instruction to JUMP, a jump of instruction is carried out and that this one points on another destination of JUMP then the mask is valid. It is impossible with our solution to detect the fault. Although this case exists, it is in fact rather difficult to

implement because it is necessary to target the precise jump cycle, but also to determine a new valid JUMP destination.

VI. Implementation

In this part, only the feasibility of implementing one of the countermeasures in the pipeline of a processor will be studied. For reasons of simplicity at compilation, it was chosen to use the method requiring only the previous instruction. However, in order to have a representative idea of the two solutions only the unmasking and mask generation has been added. The decoding verification part has not been implemented to show that it is possible to achieve a satisfactory level of security against physical attacks with a negligible overhead.

A. Hardware realization

To realize our solution, we place ourselves in the case of a 4-stage in-order RISC-V processor. Our solution is therefore added to the decode stage of the pipeline. The "*Decod_output*" register is added to store the output signals of the decoder, it is updated at each cycle except during the freeze cycles. The freeze cycles are indicated to the processor by the "*freeze_sig*" signal. In addition, the mask used for all jumps and branches is saved in the "*jump_mask*". The choice of "*jump_mask*" or "*DEC_mask*" is determined by the "*Jump_controller*". The *Jump_controller* uses the instruction

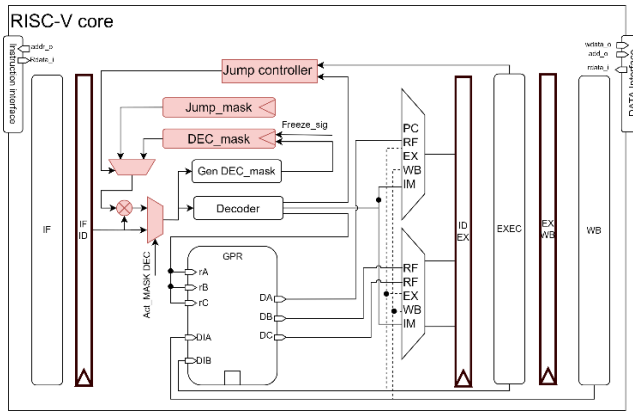


Figure 11: Integration into the CV32E40P pipeline

decoding signals and the output signals of the execution stage to determine whether the "*jump_mask*" should be used. This includes direct and indirect jump cases, but also branches taken.

The solution can be activated or deactivated using the "*ACT_DEC_mask*" signal. In the case of our implementation, this is an input signal from the processor. This signal is controlled by a register, accessible by JTAG, outside the core. This signal allows taking either the incoming instruction with the application of the mask or without the application of the mask. The problem of the first mask to be used must be solved. Indeed, the first instruction by definition has no preceding instruction. In this case, the jump mask is used. It is therefore necessary to ensure that the first instruction executed is well masked by this mask.

For more activation possibilities, a start signal can be added. This adds an attack vector as well as complexity which has not found any advantage in our use. It should be noted that this signal does not allow to deactivate the countermeasure, it only applies to the mask or not.

The two solutions are quite close to implementation, however, to have the verification of the DECOD stage with pre-decoder it is necessary to add an encoder and a mask verification, as presented Figure 10, but also to add the missing signals to reconstruct the instruction, see III.C.

We implemented our RTL architectures based on RISCY CV32E40P processors and synthesized it with the GF22FDX (GlobalFoundries 22nm FD-SOI) Standard Cells RVT library.

The overhead of this solution is quite low from a hardware point of view. Only mask register, masks applications with a *xor* and mask generation are added. The results showed that 15,295 GEs were required for the post-decoder and 17,729 GEs for pre-decoder. There are respectively 3.71% and 16.93% additional area required and an increase in total power consumption of 4.33% and 19.16% over the original RISCY core which has an area of 14,727 GEs.

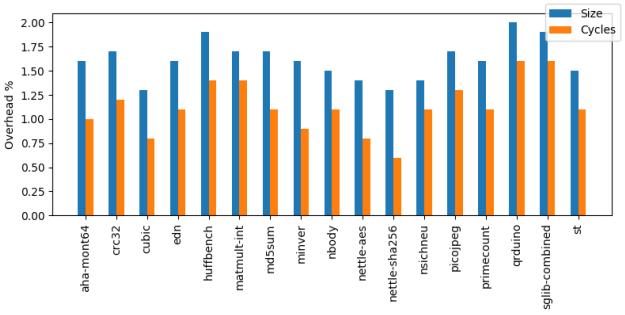


Figure 10: Overhead in execution time and code size of Embench benchmark with this solution

The figure gives the overhead of code size but also in execution time induced by the modifications made by the compiler. These results have been realized in cycle accurate simulation with the Embench1.0 benchmark. The overhead in code size is 1.61% and 1.12% for the execution time. This overhead is low because there are few modifications of compiler and instruction flow, only jump instructions are sometimes added before jump destinations.

VII. Security

The advantages of our solution are, on the one hand, its great lightness, it is enough to store 32 bits of masks and to xorize them at the incoming instruction of the next cycle. On the other hand, it has a very low overhead in terms of code size and execution time. Indeed, we only add a jump instruction before each jump destination when it is required. Moreover, the simplicity of our solution can be adapted to "just-in-time" compilations or self-modifying codes, the only difficulty being the insertion of jump instructions. Moreover, it secures against instruction jumps in an efficient way. As pointed out by A. Menu et al. [1], jumps of not one, but several instructions can occur. No matter how many instructions are skipped, our solution is still effective, unlike temporal redundancy, for example.

One of the main difficulties with countermeasures against fault injection is to measure their efficiency. Indeed, no metric allows to account for the security of a circuit in front

of all the possibilities available to the attacker. However, our attacker model being limited to the instruction path, we can assimilate all faults as a modification of the machine code. Indeed, a fault in the memory hierarchy is effectively a corruption of the binaries, and a jump of instruction can be assimilated to the suppression of its instruction. For the faults at the level of the IFETCH and DECOD stage, they can also be perceived as corruptions of the instructions. Thus, it is possible in our case to test exhaustively all the possible faults on the binaries. Because the errors propagate from cycle to cycle, we are sure that the fault will be detected in a cycle, except in case of valid jump. Thus the detection rate is not relevant but the number of cycles before a decoding error allows us to compare the different mask creation methods. With these presuppositions, an exhaustive test campaign for the detection time of a faulty instruction can be realized.

It should be noted that it is not necessary to have a hardware platform or even to execute the instructions to verify the proper functioning of our solution. Only the sequence of instructions matters. To evaluate the mask, we have developed a software tool to simulate the decoding of RISC-V instructions. To simplify the creation of the mask, we use the pre-decoder solution which only requires the instruction to generate the mask. To save computation time we execute the generated binaries a first time and then in a second time we exhaustively modify the instructions which will be effectively touched in the code. We execute 50 tests of functional verification of RISC-V architecture to ensure that all the instruction types are well used.

We have removed from our results the faults that lead to a valid jump to another jump destination. Indeed, due to the exhaustiveness of our fault injection campaign, these are inevitable cases. It is not relevant to take them into account, because they are unrealistic cases from the attacker's point of view.

1) One bit faulty

The first check is to verify that the proposed mask is sufficient for a simple fault injection. Figure 12 compare the pre-decoder solution (complete) with a 32bit mask generated with the hash function ASCON. But also with the two sub-masks which compose it, the one of the sboxes with consecutive blocks (linear_sbox) and the one with interleaved blocks (mix_sbox) see III. C. Finally, the instruction without transformation (identity).

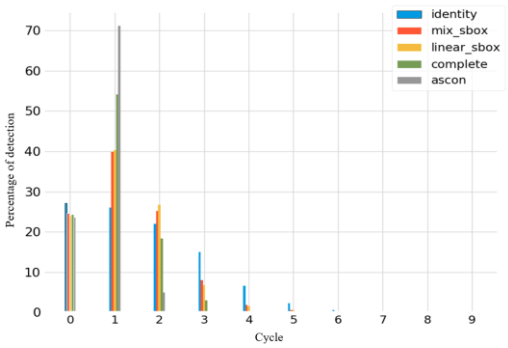


Figure 12: Mask comparison

The first thing to take into account is that the detection of cycle 0 is identical for all the types of masks, because this one does not intervene yet and in view of the exhaustiveness of our test method there is no random character. With the instruction only 35% of the remaining faults are detected in the second cycle, 45% with only 1 type of sbox, 75% of the complete mask and finally 90% with the hash generated by ASCON. Table 3 shows the average number of cycles before a fault leads to a decoding error. Thus, a hash of the instruction allows detection of the fault in 2 cycles, our solution which consists of only 16 hardware sboxes allows detecting 97% of faults within 2 cycles. Since 99% are detected in fewer than 3 cycles, we can reasonably assume that these faults have not corrupted data memory, so by disabling the pipeline and the register bank the processor is in a safe state.

2) Multiple bits faulty

Multiple faults are more and more used attacks [14] and

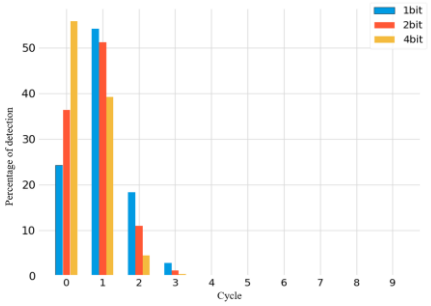


Figure 13: Multiple fault

allow to bypass the security set up against single faults. However, our solution, as can be seen in Figure 13, provides better protection as the number of faults increases. Thus an attacker has no interest in passing on more complex faults except if she wants to modify very precisely an instruction to jump to a valid jump destination, which seems precision out

	Identity	Linear_sbox	Mix_sbox	Complete	ASCON
Average cycle detection	1.52	1.20	1.23	0.99	0.81

Table 1: comparison of the average number of detection cycles

of reach of current attacks.

3) Instruction skip

Beyond the modification of instructions, it is also common that fault injections cause instruction skips that can be single or multiple, some solutions are sensitive to the number of instructions skipped. Our solution as can be seen

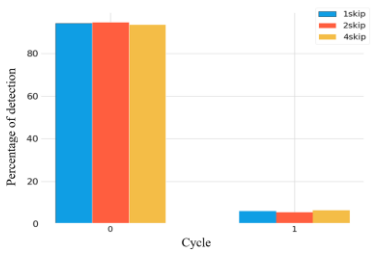


Figure 14: Instruction skip

in Figure 14 detects jumps in fewer than 2 cycles and those regardless of whether the skip is 1, 2 or 4 instructions.

As a consequence, our solution remains effective against fault attacks on the entirety of the instructions path. The more complex the attack with faults and skips of multiple instruction, the more our approach is effective.

B. Security against side channel attacks

Finally, beyond the protection against fault attacks, the fact that a mask is added to the instructions makes it possible to reinforce the resistance against attacks by side channels. However, this is not a countermeasure that formally prevents leaks by side channels, but only makes it more difficult to disassemble, which is done before the DECOD stage [3], which is to our knowledge the only attack that targets the instruction path by side channels. Even if the mask is not random, it allows reducing many heuristics such as impossible opcodes, most likely instruction sequences and dependencies between opcodes, funct3 and funct7. Without making this type of attack impossible, our countermeasure makes them much more complex.

VIII. CONCLUSION

This paper proposes two solutions to protect the instruction path against faulty and side-channel attacks. These solutions propose to generate masks either directly with the previous instructions or with the decoding signals of the previous instruction. The advantage of this type of countermeasure is that it allows to deal with all the problems of fault injection on instructions with a very low overhead. By using decoding signals we have an overhead of 3.25%, we ensure fault detection in 1 cycle on average, but makes the compilation dependent on micro architectural constraints. By taking only pre-decoder information, we ensure security against fault injection attacks on the instruction path at the cost of doubling DECOD stage. But the generated source code becomes only dependent on the instruction set.

The software overhead is very low because of the few changes made to the compiler. Indeed, we have an overhead of 1.61% in code size and 1.12% in execution time.

This solution proposes a security against the attacks by observation and physical on the instruction path, it is thus complementary to the security solution of the control flow and the data path.

ACKNOWLEDGMENT

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

REFERENCES

[1] A. Menu, J.-M. Dutertre, O. Potin, J.-B. Rigaud, et J.-L. Danger, « Experimental Analysis of the Electromagnetic Instruction Skip Fault Model », in *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, avr. 2020, p. 1-7. doi: 10.1109/DTIS48698.2020.9081261.

[2] J.-M. Dutertre, T. Riom, O. Potin, et J.-B. Rigaud, « Experimental Analysis of the Laser-Induced Instruction Skip Fault Model », in *Secure IT Systems*, Cham, 2019, p. 221-237. doi: 10.1007/978-3-030-35055-0_14.

[3] V. Cristiani, M. Lecomte, et T. Hiscock, « A Bit-Level Approach to Side Channel Based Disassembling », Prague, Czech Republic, nov. 2019. Consulté le: 6 décembre 2021.

[4] S. Sayeed, H. Marco-Gisbert, I. Ripoll, et M. Birch, « Control-Flow Integrity: Attacks and Protections », *Appl. Sci.*, vol. 9, nº 20, Art. nº 20, janv. 2019, doi: 10.3390/app9204229.

[5] R. Clercq et I. Verbauwhede, « A survey of Hardware-based Control Flow Integrity (CFI) », juin 2017.

[6] M. Werner, T. Unterluggauer, D. Schaffenrath, et S. Mangard, « Sponge-Based Control-Flow Protection for IoT Devices », in *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, avr. 2018, p. 214-226. doi: 10.1109/EuroSP.2018.00023.

[7] O. Savry, M. El-Majihi, et T. Hiscock, « Confidaent: Control FLow protection with Instruction and Data Authenticated Encryption », in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, août 2020, p. 246-253. doi: 10.1109/DSD51259.2020.00048.

[8] L. Blasi, F. Vigli, A. Cheikh, A. Mastrandrea, F. Menichelli, et M. Olivieri, « A RISC-V Fault-Tolerant Microcontroller Core Architecture Based on a Hardware Thread Full/Partial Protection and a Thread-Controlled Watch-Dog Timer », in *Applications in Electronics Pervading Industry, Environment and Society*, Cham, 2020, p. 505-511. doi: 10.1007/978-3-030-37277-4_59.

[9] N. Moro, K. Heydemann, E. Encrenaz, et B. Robisson, « Formal verification of a software countermeasure against instruction skip attacks », *J. Cryptogr. Eng.*, vol. 4, nº 3, p. 145-156, sept. 2014, doi: 10.1007/s13389-014-0077-7.

[10] A. Traber, S. Stucki, F. Zaruba, M. Gautschi, A. Pullini, et L. Benini, « PULPino: A RISC-V based single-core system »

[11] M. Gruhn et T. Müller, « On the Practicability of Cold Boot Attacks », in *2013 International Conference on Availability, Reliability and Security*, sept. 2013, p. 390-397. doi: 10.1109/ARES.2013.52.

[12] R. Schilling, M. Werner, et S. Mangard, « Securing conditional branches in the presence of fault attacks », in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, mars 2018, p. 1586-1591. doi: 10.23919/DATE.2018.8342268.

[13] K. Shibutani, T. Isobe, H. Hiwatari, A. Mitsuda, T. Akishita, et T. Shirai, « Piccolo: An Ultra-Lightweight Blockcipher », in *Cryptographic Hardware and Embedded Systems – CHES 2011*, vol. 6917, B. Preneel et T. Takagi, Éd. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, p. 342-357. doi: 10.1007/978-3-642-23951-9_23.

[14] B. Colombier *et al.*, « Multi-Spot Laser Fault Injection Setup: New Possibilities for Fault Injection Attacks », in *Smart Card Research and Advanced Applications*, vol. 13173, V. Grosso et T. Pöppelmann, Éd. Cham: Springer International Publishing, 2022, p. 151-166. doi: 10.1007/978-3-030-97348-3_9.