Formal verification of a software countermeasure against instruction skip attacks

Karine Heydemann[1], Nicolas Moro[1,2], Emmanuelle Encrenaz[1], and Bruno Robisson[2]

[1] Laboratoire d'Informatique de Paris 6 (LIP6), UPMC Univ Paris 06
{karine.heydemann,nicolas.moro,emmanuelle.encrenaz}@lip6.fr
[2] Commissariat à l'Énergie Atomique et aux Énergies Alternatives (CEA)
{nicolas.moro,bruno.robisson}@cea.fr

Extended version of the paper presented at PROOFS 2013 (Santa Barbara)

Abstract. Fault attacks against embedded circuits enabled to define many new attack paths against secure circuits. Every attack path relies on a specific fault model which defines the type of faults that the attacker can perform. On embedded processors, a fault model in which an attacker is able to skip an assembly instruction is practical and has been obtained by using several fault injection means. To handle this issue, some countermeasure schemes which rely on temporal redundancy have been proposed. Nevertheless, double fault injection in a long enough time interval is practical and can bypass those countermeasure schemes. Some fine-grained other countermeasure schemes have been proposed for specific instructions. However, to the best of our knowledge, no approach that enables to secure a generic assembly program in order to make it fault-tolerant to instruction skip attacks has been formally proven yet. In this paper, we provide a fault-tolerant replacement sequence for every instruction of the whole Thumb2 instruction set and provide a formal proof of this fault tolerance. This simple transformation enables to add a reasonably good security level to an embedded program and makes practical fault injection attacks much harder to achieve.

Keywords: microcontroller, fault attack, instruction skip, countermeasure, formal proof

1 Introduction

Physical attacks were introduced in the late 1990s as a new way to break cryptosystems. Unlike classical cryptanalysis, they use some weaknesses in the cryptosystems' implementations as a way to break them. Among them, faults attacks were introduced in 1997 by Boneh *et al.* [1]. In this class of attacks, attackers try to modify a circuit's environment in order to change its behaviour or induce faults into its computations [2]. This attack principle was first introduced against cryptographic circuits but can be used against a larger set of embedded circuits.

Many physical means can be used to induce such faults: laser shots [3] [4], overclocking [5], chip underpowering [6], temperature increase [7] or electromagnetic glitches [4] [8].

Among fault attacks, three subclasses can be distinguished: differential fault analysis, safe error and algorithm modifications. Differential fault analysis (DFA) aims at retrieving some ciphering keys by comparing correct ciphertexts with ciphertexts obtained from a faulted encryption [1] [9]. Safe-error attacks are based on the fact that a fault injection may have or not have an impact on the output [10]. Finally, algorithm modifications target an embedded processor and aim at injecting faults into an embedded program's control flow [11] [12].

Those attack schemes rely on an attacker's fault model which defines the set of faults an attacker can perform [13,14]. As a consequence, countermeasure schemes must take this fault model aspect into account. On microcontrollers and embedded processors, the fault model in which an attacker can skip an assembly instruction has been observed on different architectures [11] [15] and for different fault injection means [12] [8] [16]. As a consequence, this fault model is a realistic threat for an embedded program.

In this paper, we consider this instruction skip fault model and propose a countermeasure scheme that could enable to secure any assembly code against instruction skip faults. Some countermeasures based on multiple executions of a function have already been proposed and can theoretically handle this issue [2]. However, this kind of high granularity temporal redundancy is vulnerable to multiple fault attacks. Even with commonly-used low-cost fault injection means, a high temporal accuracy can be obtained by an attacker, and performing the same fault injection on several executions of an algorithm is practical [16]. On the contrary, performing a multiple fault on two instructions separated by a few clock cycles is significantly harder [17] while still possible. Indeed, it requires a much more costly fault injection equipment and very high synchronization capabilities. It is then not yet considered as a threat.

Our approach uses an instruction-scale temporal redundancy to ensure a fault-tolerant execution of an embedded program. It is based on the statement that performing two faults on two instructions separated by few clock cycles is hardly feasible. In this paper, we propose a fault-tolerant replacement sequence for each instruction of the whole Thumb2 instruction set. We also show how to formally prove the fault tolerance of replacement sequences by using a model-checking tool. By using such a fine-grained redundancy scheme, it is then possible to strengthen any assembly program against fault attacks without any specific knowledge about it. In the experimental results, we evaluate the overhead induced by fault tolerance and show that it can be reduced by only applying this countermeasure scheme to the sensitive parts of an implementation.

The rest of this paper is organized as follows. Section 2 introduces our fault model and gives details about some related research papers. Section 3 introduces our countermeasure scheme and details our replacement sequences. Section 4 explains the approach we use for the formal proof. Finally, section 5 evaluates the efficiency of our countermeasure scheme on an AES implementation.

2 Related works and fault model

This section is dedicated to related works. First, fault models are discussed in section 2.1. Countermeasure schemes that have previously been proposed are addressed in section 2.2. Section 2.3 presents some related research papers on formal verification.

2.1 Fault model

On embedded processors, a fault model in which an attacker can skip an assembly instruction or equivalently replace it by a `NOP` has been observed on several architectures and for several fault injection means [13]. On a 8-bit AVR microcontroller, Schmidt *et al.* [11] and Balasch *et al.* [12] obtained instruction skip effects by using clock glitches. Dehbaoui *et al.* obtained the same kind of effects on another 8-bit AVR microcontroller by using electromagnetic glitches [8]. On a 32-bit ARM9 processor, Barenghi *et al.* obtained some instruction skip effects by using voltage glitches. On a more recent 32-bit ARM Cortex-M3 processor, Trichina *et al.* were able to perform instruction skips by using laser shots [16]. Moreover, this fault model has also been used as a basis for several cryptanalytic attacks [18] [14]. As a consequence, it is considered as a common fault model an attacker may be able to perform [19].

A more generic fault model is the instruction replacement model, in which `NOP` replacements correspond to one possible case. In some previous experiments on an ARM Cortex-M3 processor by using electromagnetic glitches, we have observed a corruption of the instructions binary encodings during the bus transfers [20] leading to such instruction replacements. Actually, instruction skips correspond to specific cases of instruction replacements: replacing an instruction by another one that does not affect any useful register has the same effect as a `NOP` replacement and so is equivalent to an instruction skip. Many injection means enable to perform instruction replacement attacks [20] [12]. Nevertheless, even with very accurate fault injection means, being able to precisely control an instruction replacement is a very tough task and, to the best of our knowledge, no practical attack based on such a fault model has been published yet.

As a conclusion, we consider in this paper that an attacker is able to skip an instruction.

2.2 Countermeasure schemes

Several countermeasures schemes have been defined to protect embedded processor architectures against specific fault models. At a hardware level, many countermeasures have been proposed. As an example, Nguyen *et al.* [21] propose to use integrity checks to ensure that no instruction replacement took place.

Software-only countermeasure schemes, which aim at protecting the assembly code, are more flexible and avoid any modification of the hardware. Against fault attacks, the most common software fault detection approach relies on function-level temporal redundancy [2]. For example, this principle applied to a cryptographic implementation can be achieved by calling twice the same encryption algorithm on the same input and then comparing the outputs. For encryption algorithms, an alternative way is to call the deciphering algorithm on the output of an encryption and to compare its output with the initial input. These approaches enable fault detection and involves doubling the execution time of the algorithm. Triplication approaches with voting enabling fault tolerance at the price of tripling the execution time of the whole algorithm has also been proposed [2].

At an algorithm level, in [22], Medwed *et al.* propose a generic approach based on the use of specific algebraic structures named $AN+B$ codes. Their approach enables to protect both the control and data flow. An application to an AES implementation has also been detailed in [23].

At an assembly level, in [17], Barenghi *et al.* propose three countermeasure schemes based on instruction duplication, instruction triplication and parity checking. Their approach enables to ensure a fault detection for the *load* instructions against instruction skip or transient data corruption fault models. Our scheme enables a fault tolerance only against the instruction skip fault model but for every instruction of the whole considered instruction set. Moreover, our countermeasure scheme has been formally proven fault tolerant.

2.3 Formal verification of software countermeasures

Formal methods and formal verification tools have been used for cryptographic protocols' verification of to check that an implementation could meet the Common Criteria security specifications [24]. However, to the best of our knowledge, very few formal proof approaches to check the correctness of software countermeasure schemes against fault attacks have been proposed yet. The most significant contribution has been proposed by Christofi *et al.* [25]. Their approach aims at performing a source code level verification of the effectiveness of a countermeasure scheme on a CRT-RSA implementation by using the Frama-C program analyzer. In this paper, we formally prove all our proposed countermeasures against an instruction skip fault model at an assembly level.

3 Countermeasure scheme

The proposed countermeasure scheme aims at ensuring a fault-tolerant execution of an assembly code against instruction skip faults. The approach we propose relies on providing a formally proven fault-tolerant replacement sequence for each assembly instruction of a whole instruction set. We chose the ARM Thumb2 instruction set [26] since ARM is a widely used target architecture for embedded processors. Thumb2 is actually the successor to both ARM and Thumb instruction sets, and contains both 16-bit and 32-bit instructions. In this section, we present some of the replacement sequences we have defined for each instruction of the Thumb2 instruction set. This fine-grained redundancy scheme enables to strengthen any assembly code against fault attacks without any specific knowledge about it.

3.1 Instruction classes

We have defined a fault-tolerant replacement sequence for each instruction and each encoding of the Thumb2 instruction set. This instruction set contains 151 instructions, and each instruction has up to four different encodings. For many instructions, the replacement sequence is very simple. However, this sequence can become much more complex for some specific instructions. According to the replacement sequences found, the instructions in the Thumb2 instruction set can be divided into three classes. Every class is associated to one kind of replacement sequence. These three classes are summarized in table 1.

Table 1. Instruction classes in the Thumb2 instruction set

Instruction class	**Examples**	**Replacement scheme**

Idempotent instructions	`mov r1,r8`
	`add r3,r1,r2`
		Instruction duplication

Separable instructions	`add r1,r1,#1`
	`push {r4,r5,r6}`
		Use of extra registers and decomposition into an idempotent instruction sequence

Specific instructions	`bl <function>`
	`it` blocks
	`adcs r3,r1,r2`
		Replacement sequence specific to each instruction

The first class is composed of the idempotent instructions which only need to be duplicated to provide a fault tolerance. The second class gathers the instructions that are not idempotent but can be replaced by an equivalent sequence of idempotent instructions. The third class gathers some specific instructions that cannot easily be replaced by a list of idempotent instructions but for which a specific replacement sequence is possible. This last class also contains the instructions for which no replacement sequence that ensures a fault tolerance and a correct execution in any case can be provided. The solution for these instructions is either to avoid the compiler to use them or to use a fault detection approach. The following section gives more details about those classes. Moreover, it provides some examples of replacement sequences for every class.

3.2 Individual instruction replacement sequences

Idempotent instructions Idempotent instructions are the instructions that have the same effect when executed once or twice. If all the source operands are different from the destination operands, and if the value written into the destination operands does not depend on the instruction's location in the code, then the instruction is said to be idempotent. For such instructions, the countermeasure is a simple instruction duplication. The overhead for such a duplication is twofold: an overhead equals to the instruction size in terms of code size and a performance overhead that is equal to the time execution time for the instruction.

Table 2 gives some examples of idempotent instructions and their associated replacement sequence.

Table 2. Replacement sequences for some idempotent instructions

Instruction	**Replacement sequence**
`mov r1,r8` (copies `r8` into `r1`)	`mov r1, r8` `mov r1, r8`
`ldr r1, [r8, r2]` (loads the value at the address `r8+r2` into `r1`	`ldr r1, [r8, r2]` `ldr r1, [r8, r2]`
`str r3, [r2, #10]` (stores `r3` at the address `r2+10`)	`str r3, [r2, #10]` `str r3, [r2, #10]`
`add r3,r1,r2` (puts `r1+r2` into `r3`)	`add r3,r1,r2` `add r3,r1,r2`

Separable instructions In the considered instruction set, some instructions are not idempotent but can be replaced by a sequence of fully idempotent instructions whose execution gives the same result. Once this separation is performed, each idempotent instruction of the replacement sequence can then be duplicated. This class gathers the instructions whose destination register is also a source register. To replace these instruction by a sequence of fully idempotent instructions, an extra register has to be used. This register has to be available at this location in the code: any dead register can be used[3]. Listing 1.1 shows the replacement sequence for an `add r1, r3` instruction. For this class of instructions, the overhead cost brought by our countermeasure scheme depends on the instruction to replace. There is an overhead cost in code size, performance and register pressure (since the replacement sequence needs some extra registers). For the `add r1, r3` instruction example, one extra register is needed and 3 extra instructions, the overhead cost in terms of code size is between 6 and 10 bytes (depending on the encoding used for the `add` instruction).

Listing 1.1. Replacement sequence for the non idempotent `add r1, r3` instruction

```
; we assume rx is an
; available register
mov rx, r1
mov rx, r1
add r1, rx, r3
add r1, rx, r3
```

Listing 1.2. Replacement sequence for the `push` {`r1, r2, r3, lr`} instruction

```
; the push{} instruction
; is equivalent to the
; stmdb sp!,{} instruction
stmdb sp, {r1, r2, r3, lr}
stmdb sp, {r1, r2, r3, lr}
sub.W   rx, sp, #16
sub.W   rx, sp, #16
mov sp, rx
mov sp, rx
```

[3] It turns out that, in the ARM calling conventions, the `r12` register can be used to hold intermediate values and does not need to be saved on the stack. Thus, this register can be used as a temporary register for such replacement scenarios.

Stack manipulation instructions Some memory access instructions update the address register before or after (`stmdb`[4]/`ldmia`[5]) a memory access. As a consequence, this address register is both a source and a destination register for such an instruction. This is notably the case of the stack manipulation instructions (`push` and `pop`). These instructions respectively write or read on the stack and decrement or increment the stack pointer. Such instructions can be separated into a sequence of instructions that only perform one operation at a time, either a memory access or an address register update. The `push` instruction can be decomposed into instructions that first write the register to save on the stack and then decrement the stack pointer. As decrementing the stack pointer implies reading and writing the same register, this operation is decomposed into two steps in order to get a sequence of idempotent instructions. Such a replacement sequence for the `push` instruction is detailed on listing 1.2. This replacement requires 1 extra register and has a code size and performance overhead of 5 instructions.

Listing 1.3. Replacement sequence for `umlal rlo, rhi, rn, rm` instruction that performs `rhi:rlo = rn*rm + rhi:rlo`

```
mrs    rt , APSR    ; save
mrs    rt , APSR    ; flags
umull  rx , ry , rn , rm
umull  rx , ry , rn , rm
adds   rz , rx , rlo
adds   rz , rx , rlo
addc   rx , ry , rhi
addc   rx , ry , rhi
mov    rlo , rz
mov    rlo , rz
mov    rhi , rx
mov    rhi , rx
msr    APSR, rt   ; restore
mrs    APSR, rt   ; flags
```

`umlal` instruction The `umlal` instruction multiplies two source registers and then adds the content of the concatenation of the two 32-bit destination registers. The final result is written into the two 32-bit destination registers. As a consequence, this instruction has registers that are both source and destination. However, it can be decomposed. First, a multiply instruction whose result is a 64-bit value can be performed. Then the 64-bit addition has to be decomposed into several instructions. This requires to propagate the carry set by adding the 32 least significant bits (by using an `adds` instruction) to the addition of the 32 most significant bits by using an `adc` instruction. However the `adds` instruction sets the flags whereas the `umlal` does not: this sequence of instructions is not strictly equivalent to the `umlal` instruction and may be wrong if the flags are used after the `umlal` instruction without being set. As a consequence, it is necessary to save

[4] `stmdb` stores multiple registers into the memory in a descending direction
[5] `ldmia` loads a memory segment into multiple registers in an ascending direction

the flags before the sequence and restore them after it. Performing such a saving requires 4 extra instructions. The corresponding replacement sequence for this instruction is given in listing 1.3. This countermeasure requires 4 extra registers and replaces the initial instruction by 14 instructions. This replacement sequence is actually the most costly one of the whole instruction set, both in term of extra registers and extra instructions.

Specific instructions Some instructions cannot easily be replaced by a list of idempotent instructions. These instructions can still be decomposed into an equivalent sequence of instructions that can be duplicated to enforce a robust execution. There are also some instructions for which no fault-tolerant countermeasure in any case can be found. Some of them can still be replaced by a fault-tolerant sequence under some constraints. In this section, we give details and provide some examples for both kinds of such specific instructions.

`bl` subroutine call instruction The subroutine call instruction (`bl`) performs a jump and writes the return pointer into the link register (`r14`). Duplicating a `bl` instruction would induce two subroutine calls if no attack is performed. A possible solution is to manually put the return address into the link register and then perform an unconditional jump. As the Thumb execution mode requires the last bit of an instruction address to be set, this bit must be set before the unconditional jump to the subroutine code, as shown on listing 1.4.

Listing 1.4. Replacement sequence for a `bl <function>` instruction.

```
; Thumb mode requires
; the last bit to be set
 adr   ry,<return_label>
 adr   ry,<return_label>
 add      lr , ry , 1
 add      lr , ry , 1
 b   <function>
 b   <function>
return_label:
```

Listing 1.5. Replacement sequence for `adcs r1, r2, r3` instruction

```
; This sequence is valid
; if the flags are not alive
mrs rx , APSR  ; save
mrs rx , APSR  ; flags
adcs r1 , r2 , r3
msr APSR, rx   ; restore
msr APSR, rx   ; flags
adcs r1 , r2 , r3
```

Instructions that both read and write the flags Instructions that read and write the flags cannot easily be replaced by a fault-tolerant sequence of instructions. For example, the `adsc` instruction performs an addition between two source operands (two registers or one register and an immediate value) and the carry flag. The result is written into a destination register and the flags (carry, negative, overflow and zero) are updated. Duplicating such an instruction is not correct since the second `adcs` would use the carry set by the first `adcs` instruction instead of the initial carry value. If the flags are read before being written whatever the execution path after the `adcs` instruction is, then no simple replacement sequence is possible, the code has to be modified. Otherwise, if the flags are written before being used again, a replacement sequence is possible. Such a sequence consists in saving the flags values before the first `adcs` instruction and restoring these values before the second `adcs` instruction. This replacement sequence is illustrated in listing 1.5.

Listing 1.6. Example of `it` block

```
  itte NE
  addne r1, r2, 10
  eorne r3, r5, r1
  moveq r3, #10
```

Listing 1.7. Code equivalent to the `it` block above

```
  b.eq else
  add r1, r2, 10
  eor r3, r5, r1
  b continuation
else
  mov r3, #10
continuation
```

Listing 1.8. Code of listing 1.7 strengthened with our individual instruction countermeasure scheme

```
  b.eq else
  b.eq else
  add r1, r2, 10
  add r1, r2, 10
  eor r3, r5, r1
  eor r3, r5, r1
  b continuation
  b continuation
else
  mov r3, #10
  mov r3, #10
continuation
```

`it` blocks Thumb2 provides conditional execution of instructions through `it` blocks. An `it` instruction specifies a condition and up to the 4 following instructions can be conditionally executed according to this condition or its inverse. `it` blocks correspond to if-then or if-then-else higher-level constructions and are useful when the branches of a conditional statement are composed of a limited number of instructions. Listing 1.6 gives an example of such an `it` block. The simplest solution for such blocks is to first transform the `it` block into an equivalent classical if-then-else structure such as the one presented on listing 1.8 and then apply the countermeasure scheme to each instruction, as illustrated on listing 1.7. However, we have defined a specific replacement sequence for `it` blocks but this replacement has some limitations and can quickly become more costly than its equivalent form with an if-then-else structure. Due to the lack of space, this replacement sequence is presented in appendix **??**.

4 Formal proof of fault tolerance

In this section, we present how we formally prove the fault tolerance specification for the countermeasure replacement sequences we presented in section 3. Details about the modelings we use for the proof approach are presented in 4.1 and proof examples for some replacement sequences are presented in 4.2.

4.1 State machine modeling and specification to prove

A program acts as the application of transformations of the values stored in the set of registers or in memory. Each instruction of the program acts like a function whose input is a registers and memory configuration and produces a new registers and memory configuration. The program can be represented as a transition system whose states are registers and memory configurations and transitions mimic the state transformation applied by the instructions.

Individual instruction modeling Instead of proving the fault tolerance for a complete program, our model checking approach consists in proving the fault tolerance for each replacement sequence proposed in our countermeasure scheme. Indeed, it is sufficient to certify that the output state (register and memory configuration) after the replacement sequence execution (with or without a fault injection) is equivalent to the normal output state after the initial instruction execution. As this output state is also the input state for the following instruction, using such a proof approach certifies that the next instruction will start from the right configuration. Moreover, this enables to use model checking while avoiding state-explosion problem.

State machine modeling We can model the execution of a sequence of instructions by a transition system TS. We define this transition system as $TS = \{S, T, S_0, S_f, L\}$. S is the set of states, T the set of transitions $T : S \rightarrow S$, S_0 and S_f are the subsets of S which respectively gather the initial states and final states. The final states from S_f are absorbing states. A state from S is defined by the value of the different registers (from the set of registers R which includes the program counter) and processor flags (from the set of flags F). Each transition from T is defined by the effect of an instruction on the registers and processor flags. L is a set of labels which correspond to the values the program counter can take. An example of such a transition system for the `add r1, r2, r3]` instruction is shown in Fig. 1. To prove that a countermeasure for an instruction i is robust against a fault, we build two transition systems: a first one for the initial instruction $m(i)$ and another one for its strengthened replacement sequence $m_{cm}(i)$.

```
# input r2, r3, flags
# output r1, flags
pc_init  : add r1, r2, r3
pc_final : next_instruction
```

$pc\_init, pc\_final \in L$
(R, F) is the current state
(R', F') is the next state
$t : (R, F) \rightarrow (R', F')$ with
$R.pc = pc\_init$
$R'.r1 = R.r1 + R.r2$
$R'.pc = pc\_final$

Fig. 1. Transition system for the `add r1, r2, r3` instruction

Fault modeling In any transition system $m_{cm}(i)$, skip fault or data transient fault may occur. An instruction skip fault is modeled by a transition from a state to any following one. Such a faulty transition only modifies the program counter. We add to the whole transition system a skip instruction faulty transition between every pair of adjacent states. As we assume that only one skip instruction fault injection may occur, every fault transition is guarded with a boolean which identifies that a fault has already occurred.

Flags and registers modeling The set of registers is composed of the general-purpose registers (`r0-r12`), the stack pointer (`r13`), the link register (`r14`) and the program counter (`r15`). The 5 processor flags are: C (carry), N (negative), Z (zero), V (overflow), Q (saturation). These flags can be set by some instructions and are used by several others. The conditional jumps are among the instructions that use those flags. Each flag is modeled as a 1-bit register. All the other registers are modeled as 4-bit registers. This width is sufficient to model the arithmetic and logic operations as well as the flags computations and enables to keep a reasonable complexity for the model checker. Moreover, modeling all the registers is not necessary since an instruction only reads a subset of the registers and writes on the destination registers. Besides, according to our fault model, the registers that are not modified by an instruction cannot be modified by a fault. Thus, for a given $m(i)$ or $m_{cm}(i)$, the set of registers R is only composed of the subset of registers that are manipulated by i or its replacement sequence $cm(i)$. Newly introduced registers in $cm(i)$ are supposed to be dead after the occurrence of the instruction i in the initial program.

Memory modeling Since in our fault model we assume the memory cannot be corrupted, modeling the memory is not relevant. To ensure that a write to the memory took place, we only need to ensure that the corresponding instruction has been executed at least once. As explained later in this section, we add a counter variable to $m(i)$ and $m_{cm}(i)$ in order to achieve this. For the loads from the memory, we use symbolic values as the values cannot be corrupted and they also do not matter since the formal proof consists in checking the equivalence for any value. The important point is to give the same symbolic value to any loads at a given address for the transition system $m(i)$ (when i is a *load* instruction) and for $m_{cm}(i)$. This is achieved by adding a variable for each memory address that is read by i and $cm(i)$ to both transition systems. These variables contain the needed symbolic values.

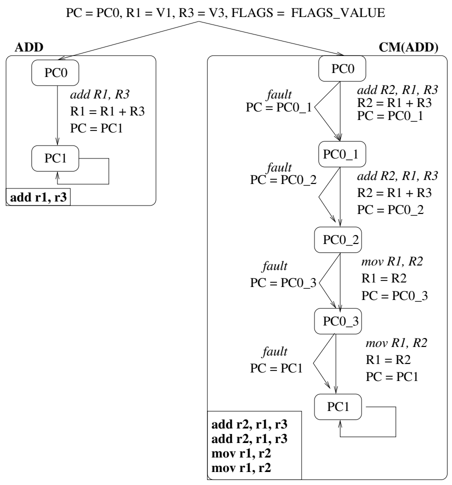
Vis model checker We used the Vis model checker[6] to prove the fault tolerance of our countermeasure scheme. This tool can take as input a transition system described with a subset of the Verilog hardware description language. Using Verilog is convenient to model transition systems which manipulate registers and bit vectors. The Vis model checker supports symbolic model checking techniques which enable to perform the proof in a symbolic way without having to enumerate each value for the registers. The proofs presented in this paper required less than one second to compute.

Specification to prove To prove the equivalence of the output of an instruction and its replacement sequence, we prove the validity of logic formulas on the two modelings. To perform such a proof, we use a specific construction in which the two transition systems $m(i)$ and $m_{cm}(i)$ have the same values for the set of registers R (except for the program counter), the set of flags F and the symbolic

[6] http://vlsi.colorado.edu/~vis/

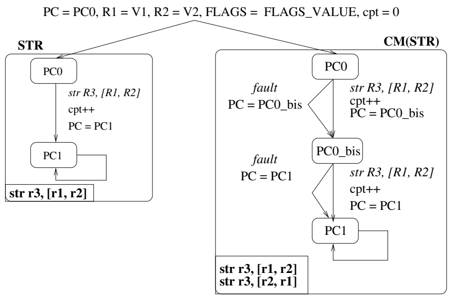
values (for the memory loads) in their initial states. Such constructions are presented in Fig. 2, 3 and 4. We need to prove that $m(i)$ and $m_{cm}(i)$ always reach a final absorbing state. Moreover, we also need to prove that, when $m(i)$ and $m_{cm}(i)$ reach a final state, the values for the set of alive registers R' (except for the program counter) and flags F' are similar. Such properties to check are expressed with the CTL temporal logic.

4.2 Formal proof of fault tolerance for some replacement sequences



```
P1: AF(ADD.PC = PC1)
P2: AF(CM(ADD).PC = PC1)
P3: AG(((ADD.PC=PC1)*(CM(ADD).PC=PC1)) =>
     ADD.R1 = CM(ADD).R1 &
     ADD.FLAGS = CM(ADD).FLAGS)
```

Fig. 2. Modeling one non idempotent instance of the `add` instruction and its countermeasure



```
P1 : AF(STR.PC=PC1)
P2 : AF(CM(STR).PC=PC1)
P3 : AG((STR.PC=PC1 * CM(STR).PC=PC1) =>
     (CM(STR).cpt = 2 + CM(STR).cpt = 1))
```

Fig. 3. Modeling one idempotent instance of the `str` instruction and its countermeasure

Idempotent and separable instructions The left part of Fig. 2 shows the state machine corresponding to the transition system for a non-idempotent `add r1,r3` instruction. The program counter is updated and depending on the instruction, the registers or the flags may be updated too. The replacement sequence uses a dead register `r2` and two extra `mov` instructions to write the result to the destination register `r1`. Its transition system is modeled by the state machine on the right part of Fig. 2. To prove that the replacement sequence is fault tolerant against a possible instruction skip, both state machines are fed with the same values for the source registers (`r1` and `r3`) and flags. Then, the validity of three CTL logic formulas has been checked with the Vis model checker. P1 and P2 express the fact that in both state machines any path from an initial state goes to a final state. P3 expresses the fact that in this final state, for all possible values in the source registers, the values in `r1` and the flags are identical in $m(i)$ and $m_{cm}(i)$. Fig. 3 presents the transition systems for an idempotent memory write, an `str r3, [r1, r2]` instruction, and its replacement sequence. In this case, as the instruction writes the content of `r3` to the memory at the address `r1`+`r2`, no proof is needed on the value inside the registers. We only need to make sure that at least one `str` instruction has been executed. A counter variable is added to the definition of a state. This counter is set to 0 and is incremented by any transition which corresponds to a `str` instruction. P1 and P2 express the fact that any path goes to the last state. P3 expresses the fact that the number of writes made by the replacement sequence greater or equal to the number of writes made by the initial instruction (which is equal to 1).

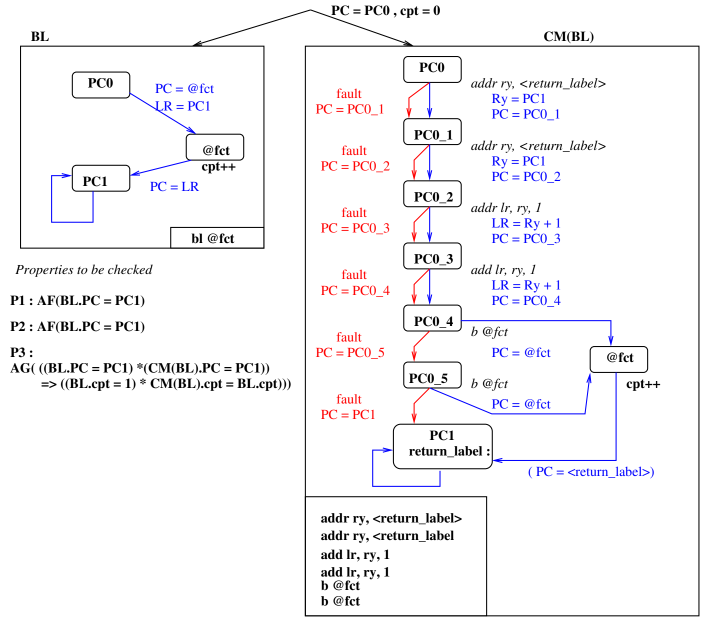


Fig. 4. Transition systems for the `bl` instruction and its replacement sequence

Specific instructions

Subroutine call: the `bl` instruction Figure 4 shows the state machines for the `bl` instruction and its replacement sequence. In both corresponding transition systems, we have added a label $@fct$ to model the target of the subroutine call. Transitions from a state in which $PC = @fct$ assign the link register to the PC. Such a transition models the return of the function and also increments a counter. Then, properties P1 and P2 to be checked by the model checker express that any path from an initial state goes to a final state. Property P3 expresses the fact that in a final state the number of calls to the function (the counter values) are the same. Validity of property P3 ensures that the function has been executed only once while validity of P1 and P2 ensures that the control flow comes back to the calling function.

Instructions that read and write the flags For the `adcs` instruction and its replacement sequence, as presented in listing 1.5, the CTL properties are the same as the ones that were used for the `add` instruction. However, the property that deals with the equality of the destination register and the flags is not valid if a fault targets the last `adcs` instruction. Relaxing the constraint on flags equality (expressed as `LIGHT_RESULT` below) makes this property valid as shown with the output of the Vis model checker in Fig. 5. To sum up, this countermeasure can only be used if the flags are not used before being set again after the `adcs` instruction.

```
MC: formula passed - AG(AF(adcs.pc=PC1))
MC: formula passed - AG(AF(cm(adcs).pc=PC1))
MC: formula passed - AG(((adcs.pc=PC1*cm(adcs).pc=PC1)->LIGHT_RESULT=1))
MC: formula failed - AG(((adcs.pc=PC1*cm(adcs).pc=PC1)->RESULT=1))
```

Fig. 5. VIS Model Checker output for the equivalence checking of the `adcs` instruction

5 Application to an AES implementation

In this section, we applied our countermeasure scheme to an implementation of the AES-128 symmetric encryption algorithm. In our implementation, every round key is calculated before the associated `AddRoundKey` operation. We provide an estimation of the overhead cost brought by our countermeasure scheme and perform an exhaustive instruction skip simulation on an ARM Cortex-M3 microcontroller to confirm the effectiveness of our approach. The chosen target is an up-to-date 32-bit microcontroller based on the ARM Cortex-M3 processor [27]. This microcontroller uses an ARMv7-M Harvard architecture and runs the Thumb2 instruction set [26].

Estimation of the overhead cost The overhead cost in terms of clock cycles and code size for two implementations that use our countermeasure scheme is shown on table 3. In the first implementation, the whole code has been strengthened with our methodology. Both overhead costs are high with this implementation. Another approach consists in applying our countermeasure to the last two rounds. In terms of cryptanalysis, fault injections are supposed to be harder to exploit if the fault does not target the last two rounds. This last scenario is just an example of a possible optimization. It enables to reduce the clock cycles overhead by strengthening some specific parts of an algorithm but some cryptanalysis attacks may still exist against such an implementation.

The overhead cost brought by our countermeasure scheme is high, but remains comparable to the one brought by classical algorithm triplication or other software approaches for fault tolerance. However, unlike such classical algorithm duplication or triplication approaches, our countermeasure scheme should be resistant to double fault attacks in a time interval longer than a few clock cycles.

Table 3. Countermeasures overhead for an AES implementation

	Clock cycles	**Relative increase**	**Code size**	**Relative increase**
AES - without countermeasure	9595		490 bytes	
AES - whole code with CM	20503	113.7 %	1480 bytes	202 %
AES - last two rounds with CM	11374	18.6 %	1874 bytes	282.5 %

6 Conclusion

In this paper, we have presented a countermeasure scheme that enables to strengthen an embedded program and make it tolerant to instruction skip faults. In our countermeasure scheme, we have build a fault-tolerant replacement sequence for each instruction of the whole Thumb2 instruction set. The instructions can be separated into three classes, which all have their dedicated replacement sequences. We have also provided a formal proof in order to guarantee the correctness and the fault tolerance of our replacement sequences for each class of instructions.

Finally, we do not claim our scheme enables a full protection against fault attacks. Nevertheless, such an approach enables to add a reasonably good security level to an embedded program, without requiring any extra hardware countermeasure and any specific knowledge about the embedded program. The overhead cost brought by using such a countermeasure is comparable to the extra cost brought by using classical algorithm-level temporal redundancy approaches and can be reduced with a more accurate knowledge about the sensitive parts that should be protected. Moreover, using a very fine-grained redundancy at the instruction scale makes the multiple fault attacks less practical with a reasonable cost equipment.

In future works, we will try to extend the fault model to a global bus corruption fault model in which data loads from the memory can also be corrupted. This fault model extension will require some changes in our fault tolerance proofs. Furthermore, we also aim at performing a practical evaluation of our countermeasure scheme by trying to attack a secured implementation on a real microcontroller with real fault injection means.

References

1. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults. Journal of Cryptology **1233** (1997)
2. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE (February 2006)
3. Skorobogatov, S.P., Anderson, R.J.: Optical Fault Induction Attacks. Cryptographic Hardware and Embedded Systems - CHES 2002 **2523**(August) (2003)
4. Schmidt, J.M., Hutter, M.: Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results. In: Austrochip 2007, Graz, Austria

5. Agoyan, M., Dutertre, J.m., Naccache, D., Robisson, B., Tria, A.: When Clocks Fail: On Critical Paths and Clock Faults. In: CARDIS 2010. (2010)
6. Zussa, L., Dutertre, J.m., Clédière, J., Robisson, B., Tria, A.: Investigation of timing constraints violation as a fault injection means. In: DCIS 2012
7. Skorobogatov, S.: Local Heating Attacks on Flash Memory Devices. In: IEEE International Workshop on Hardware-Oriented Security and Trust - HOST'09. (2009)
8. Dehbaoui, A., Dutertre, J.M., Robisson, B., Tria, A.: Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES. FDTC 2012
9. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: CRYPTO'1997, Santa Barbara, California, USA (1997)
10. Yen, S., Joye, M.: Checking before output may not be enough against fault-based cryptanalysis. Computers, IEEE Transactions on **49** (2000)
11. Schmidt, J.M., Herbst, C.: A Practical Fault Attack on Square and Multiply. In: FDTC 2008, IEEE (August 2008)
12. Balasch, J., Gierlichs, B., Verbauwhede, I.: An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In: FDTC 2011, IEEE
13. Verbauwhede, I., Karaklajic, D., Schmidt, J.M.: The Fault Attack Jungle - A Classification Model to Guide You. In: FDTC 2011, IEEE (2011)
14. Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. Proceedings of the IEEE **100**(11) (November 2012) 3056–3076
15. Barenghi, A., Bertoni, G.M., Breveglieri, L., Pelosi, G.: A fault induction technique based on voltage underfeeding with application to attacks against aes and rsa. J. Syst. Softw. **86**(7) (July 2013) 1864–1878
16. Trichina, E., Korkikyan, R.: Multi Fault Laser Attacks on Protected CRT-RSA. In: 2010 Workshop on Fault Diagnosis and Tolerance in Cryptography, IEEE (2010)
17. Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., Regazzoni, F.: Countermeasures against fault attacks on software implemented AES. In: Proceedings of the 5th Workshop on Embedded Systems Security - WESS'10, New York, USA (2010)
18. Schmidt, J.M., Medwed, M.: A Fault Attack on ECDSA. In: FDTC 2009, IEEE
19. Karaklajić, D., Schmidt, J.M., Verbauwhede, I.: Hardware Designer's Guide to Fault Attacks. IEEE Transactions on Very Large Scale Integration Systems (2013)
20. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In: FDTC 2013, Santa Barbara, California, USA (2013)
21. Nguyen, M.H., Robisson, B., Agoyan, M., Drach, N.: Low-cost recovery for the code integrity protection in secure embedded processors. In: 2011 IEEE International Symposium on Hardware-Oriented Security and Trust, IEEE (June 2011) 99–104
22. Medwed, M., Schmidt, J.M.: A Generic Fault Countermeasure Providing Data and Program Flow Integrity. In: FDTC 2008, IEEE (2008)
23. Medwed, M.: A Continuous Fault Countermeasure for AES Providing a Constant Error Detection Rate. In: FDTC 2010, IEEE (2010)
24. Chetali, B., Nguyen, Q.h.: Industrial Use of Formal Methods for a High-Level Security Evaluation. In: FM 2008: Formal Methods. (2008) 198–213
25. Christofi, M., Chetali, B., Goubin, L., Vigilant, D.: Formal verification of a CRT-RSA implementation against fault attacks. Journal of Crypt. Eng. (2013)
26. ARM: ARM Architecture Reference Manual - Thumb-2 Supplement (2005)
27. Yiu, J.: The Definitive Guide To The ARM Cortex-M3. Elsevier Science (2009)