Efficient Design and Evaluation of Countermeasures against Fault Attacks using Formal Verification

Lucien Goubet[1], Karine Heydemann[1], Emmanuelle Encrenaz[1] and Ronald De Keulenaer[2]

[1] Laboratoire d'Informatique de Paris 6 (LIP6), UPMC Univ Paris 06, France
{lucien.goubet, karine.heydemann, emmanuelle.encrenaz}@lip6.fr
[2] Computer Systems Lab (CSL), Ghent University, Belgium
ronald.dekeulenaer@elis.ugent.be

Abstract. This paper presents a formal verification framework and tool that evaluates the robustness of software countermeasures against fault-injection attacks. By modeling reference assembly code and its protected variant as automata, the framework can generate a set of equations for an SMT solver, the solutions of which represent possible attack paths. Using the tool we developed, we evaluated the robustness of state-of-the-art countermeasures against fault injection attacks. Based on insights gathered from this evaluation, we analyze any remaining weaknesses and propose applications of these countermeasures that are more robust.

Keywords: fault attack, countermeasure, formal proof

1 Introduction

More and more embedded systems, widely used in our everyday lives, hold information that is both personal and confidential (e.g., smartphones, IoT devices, passports, credit cards and SIM cards). These systems are subject to physical attacks, among which are fault attacks, that aim at disrupting the execution of programs running on a system to further an attacker's personal gain. There exist various means for injecting faults, such as electromagnetic or laser radiation, power or clock signal tampering, etc. [1, 2]. By causing a fault with a specific effect, an attacker can cause sensitive information to be leaked. For example, it has been proven that the well-known RSA encryption algorithm can be broken by differential fault analysis [3, 4]. An attacker may also take control by interfering with the boot process, bypass protections to gain access to a service running on a device, or overflow a buffer during a subroutine which is generating output, which may cause leakage of sensitive personal data. Fault injection attacks can also aid an attacker in subverting countermeasures against Simple Power Analysis [5], thus allowing him to perform side-channel analysis of a program.

Many countermeasures[3] have been proposed to prevent faults from modifying a program's execution, both in hardware and in software. In any case, to provide maximum security it is necessary to combine hardware and software countermeasures. Software countermeasures have the advantage of not requiring any hardware to be manufactured again in order to provide a stronger protection. Moreover, industries such as smart card industries and mobile phone manufacturers often rely on pre-existing hardware on top of which they have to build software security solutions.

Software countermeasures can be designed at different levels, such as at an algorithmic level [6], in a high-level programming language [7–9] or at assembly level [10–12]. While higher level countermeasures may be optimized away or altered by a compiler, low-level countermeasures are compatible with existing compilers and toolchains. Also, they allow a finer study of protections since they are closer to the final code running on the chip and thus to the effect of a physical attack. Protections are designed with respect to a fault model describing a set of effects a fault can have at a certain level of abstraction [13]. For example, two well-known fault models that describe a fault at a logical level are Single Event Upset (SEU) and Multiple Event Upset (MEU). Examples of fault models that describe a fault at assembly level include instruction skip (the execution of a single instruction is skipped), instruction replacement (the execution of a single instruction is replaced by the execution of another instruction), conditional jump inversion, jump (modification of the program counter), etc.

Software countermeasures often rely on adding code and thus have an impact on code size (memory footprint) and performance (number of executed instructions) of programs. This has important consequences for both security specialists designing countermeasures and software designers applying them to their software: they both aim to maximize security while minimizing overhead. Also, it is of the utmost importance that the application of a countermeasure effectively protects code the way it is intended. However, for security specialists designing countermeasures, it is difficult to take into account every possible context in which a certain type of fault can occur, and to predict every possible effect a fault may have. Depending on the fault model taken into consideration, design complexity increases exponentially in relation to the number of instructions that have to be protected and the number of possible control flow transfers (e.g. calls or conditional branches) in the code. To truly guarantee that a software countermeasure works correctly, a formal proof is required, just as it is required to truly guarantee that a program functions the way it was intended. In practice, formally proving the correctness of software countermeasures is done only by few experts, and it is very time-consuming.

In this paper we propose a formal setting and an automated environment to check and evaluate the robustness of software countermeasures against faults by examining code fragments representing applications of those countermeasures. Using the formal framework, it is possible to guarantee that the application

[3] Throughout this manuscript, we will use the terms *countermeasure(s)* and *protection(s)* interchangeably

of a countermeasure on a reference code fragment is correct and robust with respect to a given fault model. It also allows to aid in the design of new software countermeasures by exhibiting weaknesses, and can therefore help developers to test and deploy countermeasures more quickly. We illustrate the framework's use by evaluating the robustness of state-of-the-art protections, and we show how it can aid in the design of effective countermeasures.

The remainder of this paper is structured as follows: first, Section 2 discusses related work. The formal framework and the corresponding tool are presented in Section 3. Section 4 evaluates the robustness of a number of well-known existing countermeasures. In this section, we will also show how our tool can help enforce an existing countermeasure to be more robust. Finally, Section 5 draws conclusions.

2 Related Work

Many works have proposed software countermeasures against physical fault attacks. Software countermeasures are often based on temporal redundancy (i.e. performing the same computation multiple times) to detect or tolerate errors during computations [10, 12, 8, 11]. Control flow protection requires different mechanisms to detect a modification of the execution flow [7, 14]. A generic and automatic protection scheme for control flow integrity at C level has been proposed in [7]. The major drawback of this approach is its distance to the machine code: some faults, occurring at assembly level, may be impossible to model at source level due to the gap between the granularity of the fault at low level (one instruction) and at source level (one C statement). Moreover, a source-level protection may be removed by an optimizing compiler. A verification step at assembly level is necessary.

Barenghi et. al. propose assembly-level countermeasures based on software redundancy and parity checking to detect instruction skips [10]. The proposed countermeasure scheme, based on instruction duplication and triplication, is claimed to be robust against any single instruction skip fault. However, it becomes difficult to determine the effectiveness of this protection against other fault models – and, more largely, of any protections on large code – without the help of formal methods. There are different means to prove (security) properties: model checking [12], SAT [15], SMT [16], taint analysis [17], rewriting rules using modular arithmetic [6], use of a proof assistant like Coq [18], etc. Moro et al. have proposed countermeasures and proved their tolerance against an instruction skip using model checking with BDD [12]. In [7] the model checking approach was used to design the generic protection scheme, without which it would have been difficult, if not impossible, to elaborate a protection that defends against all attacks for the considered fault model. However, model checking with BDD does not allow representation of larger problems, since this technique faces combinatorial explosion. Other formal methods like SAT/SMT do allow larger problems to be modeled without requiring an unreasonable amount of time for verification.

Bayrak et. al. have proposed a SAT-based tool to determine which instruction of a Boolean program is sensitive to power-analysis, according to a Hamming weight uni-variate leakage model. Eldib et al. later proposed a SMT-based technique to automatically build perfectly masked Boolean programs [16]. Both methods target side channel attacks and are limited to specific assembly codes.

In his thesis [19], Moro showed that a significant percentage of faults induced by electromagnetic waves can be modeled as instruction replacements (with regard to the attacked microcontroller). This fault model remains rarely treated by countermeasures in literature, despite the fact that it can describe a large group of faults. While the countermeasure proposed by Barenghi et. al. [10] should be able to detect some instruction replacement faults, the exact types of replacements are not analyzed.

Fault attacks are a powerful means to break security. Despite the need for assistance in countermeasure design, to the best of our knowledge, no study using a formal method to evaluate the robustness of assembly countermeasures against transient faults inducing instruction replacement has ever been proposed.

3 Robustness Evaluation Framework

In order to analyze the robustness of a hardened snippet of code, our framework takes two inputs: one piece of reference code, which represents a fragment of assembly code without any protection, and a hardened piece of code to be compared to the reference code. Both faulted and non faulted executions of the hardened code are considered, and compared with non faulted execution of the reference code for robustness evaluation. In the representation of the protected code, locations at which faults may occur are given, as well as the types of faults that may occur during execution at each of these locations, and the type of robustness that is globally wanted. The robustness type is either *fault tolerance* or *fault detection*. The notion 'type of fault' refers to fault models the framework is able to consider. Currently, instruction skip and a restricted version of instruction replacement (detailed in Section 3.1) are available.

Figure 1 shows an overview of our approach. From the inputs described above, the framework constructs a set of logical predicates whose satisfiability, which can be determined with the help of a SMT solver, answers the question: "Is the protected code robust against faults occurring during execution at the specified locations?" The framework internally represents code as interpreted automata and adds transitions that correspond to the effect of injected faults to the automaton representing hardened code. Robustness evaluation is then expressed as a logical property referring to the unfolded automata which represent the execution paths of the bounded execution of each automaton.

This process results in a SMT formulation of the robustness evaluation of hardened code with respect to its corresponding reference code. The results given by a SMT-solver determine whether the hardened code is robust, and can also be interpreted to understand any vulnerabilities still present in the code.

The following subsections detail the formal models that are used, the proof scheme, and its representation with a propositional formula whose satisfiability is checked.

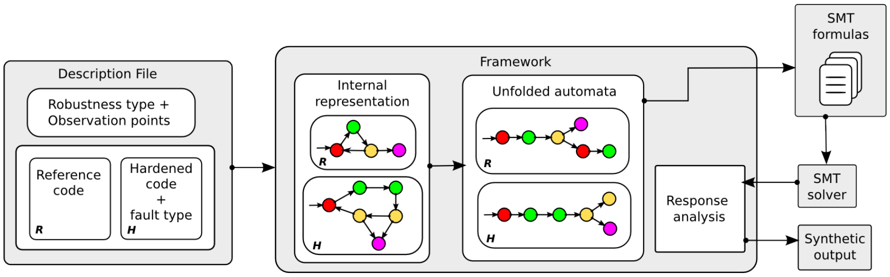

Fig. 1. Global overview

3.1 Representation of Code with Automata

The assembly code we take into consideration is ARM Thumb2 assembly (although basic operations are common to many assembly languages). Thus, the registers correspond to all user registers, among which there are the program counter, the stack pointer, the link register, and a conditional bit-wise register called `flags` that keeps the status flags. Although registers have a width of 32 bits, we have built our framework to make it possible to limit the width of registers to allow results to be calculated faster.

One way to check a property on a sequence of instructions is to model that sequence by an interpreted finite automaton. An interpreted finite automaton interferes with an external data domain by means of guarded-command transitions: the values of the data domain can restrict the firability of the transitions labelled by a guard (a Boolean expression over the data). The transition firing can modify the values of the data, by applying the action labelling the transition. In our case, each code excerpt is represented by such an interpreted automaton, interacting with a data domain (let's call it $\mathbb{D}$) containing variables representing registers (`R0` – `R15`, `flags`) and addressed memory locations. Each state of the automaton refers to a position of the program counter in the code, and each transition is labelled with a couple (guard, action), representing the corresponding assembly instruction of the code. The guard establishes the data condition associated to the firability of the transition (the flag condition in case of a conditional branch for instance). The action models the effect of the execution of the instruction on the data variables. Moreover, each transition is labeled with the set of registers that are alive after that transition. This information is used to determine the attack that may affect program execution if launched at this point during execution. The representation of assembly code as an automaton, as well as the information on register liveness, can be produced by a compiler.

Let's consider a simple load from memory and its protected version with duplication and detection, given on page 12 in Listing 7 and Listing 8 respectively. A simplified representation of the automata that correspond to both code fragments are given in Figure 2.

Fault Injection Faults are represented by added transitions whose effect on variables depends on the fault model. Currently, two fault models can be specified : *Instruction Skip* and *Simplified Instruction Replacement*, respectively denoted as IS and SIR throughout the remainder of this text. In the IS fault model, the instruction may be skipped. When this happens, the data variables are not modified, but the program counter is set to the address of the next instruction. This can be seen as executing a `nop` instruction. The instruction skip fault is represented by a single transition which does not affect any register values. In the SIR fault model, an (original) instruction may be replaced by another instruction. We restrict ourselves to the case where the replacing instruction can not directly affect either the memory nor the normal control-flow. Thus, it can not be a store into memory or a jump[4]. Although this fault model does not model all possible instruction replacements, it allows to cover a significant set of possible attacks. Moreover, understanding how to build countermeasures against this simplified fault model is the first step in designing robust countermeasures against "full" instruction replacement. This fault model is represented by a set of transitions; each of them affects one register in the set of live output registers of the associated instruction. In Figure 2, transitions that represent faults are drawn as dotted lines. In this example, every instruction of the countermeasure can be skipped (except for the second `ldr` instruction, which can be affected by the SIR fault model).

For this paper, we expect a fault to occur at most once during the execution of the hardened code snippet. However, the formal model can be easily modified to accommodate any (bounded) number of faults, at the expense of increased verification time. The number of faults restrict the firability of fault transitions: in case of one single fault, a boolean variable is added and set once a fault occurred, and all faulty transitions are guarded with a condition specifying that no fault previously occurred. Using a counter for the condition would allow to model the occurrence of several faults.

3.2 Robustness Proof Scheme

The robustness of the hardened code is formally assessed by comparing the sets of behaviors of the reference and hardened code fragments. This comparison is performed by checking dedicated properties on the set of bounded execution traces produced by the automata of both fragments (A_o and A_c for the automaton of the original code and that of the hardened code, respectively).

[4] Seeing as a fault can, however, corrupt flags and/or register contents, control-flow and memory content can be affected indirectly.

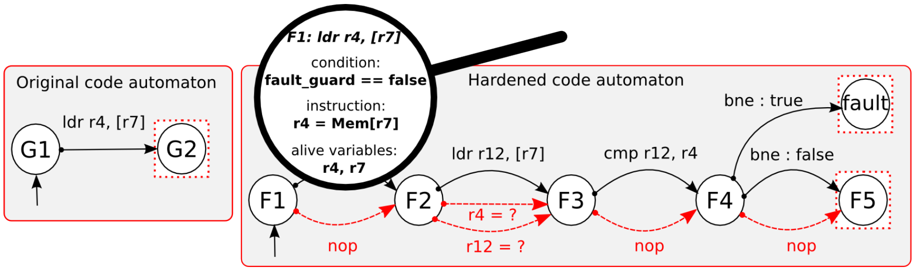

Fig. 2. Proof scheme: Automata of Listing 8 and its unhardened code

The properties to be verified depend on the type of robustness under consideration: in case of tolerance, one must ensure that, for any fault occurring once in any place of A_c, at given observation points, a set of variables $\mathbb{S} \subseteq \mathbb{D}$ specified by the designer have exactly the same value at these points; in case of detection, one must ensure that, for any fault occurring once in any place of A_c, either no fault is detected and the two automata end their execution in correct observation points with identical values in all variables of $\mathbb{S}$, or A_c ends in a fault detection observation point. Those states of both automata that are observation points are specified by designer. In most occasions we encountered, the observation points are the final or fault detection states of both automata. An example is given in Figure 2, where observations points are surrounded by dotted squares.

The verification of these properties is performed in three steps: 1) Automata unfolding, 2) Automata combination and property construction, 3) Satisfiability check with the help of a SMT solver and counter-example analysis.

SMT Representation of Automata: Unfolding From the internal representation of automata, the framework computes all possible paths from initial states to the deepest observation points. In other words, the automata are *unfolded*. In case of loops, some bounds are given to the framework to make it able to compute these paths. As an example, Figure 3 represents the unfolded automaton of the instruction duplication countermeasure in Listing 8, along 5 steps. Notice that, because of the condition on transitions, not all paths can be taken. As an example, paths involving two faulty transitions are not possible, and thus our framework will not consider them as possible attack paths. On each possible path, a transition, faulty or not, corresponds to one execution step. The beginning of the path is at step 0 and each transition on the computed execution paths is numbered. Using the execution paths and this numbering, the framework derives a SMT formulation describing all possible evaluations of the registers, flags and memory locations (variables of $\mathbb{D}$) from an initial state.

To represent the evolution of a variable, we must declare as many logical variables as the number of the unfolding steps: a variable from a certain step i

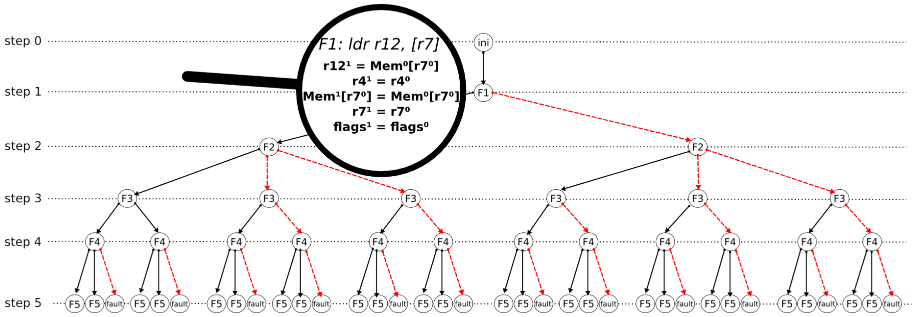

Fig. 3. Unfolded automaton of Listing 8

is calculated with the variables at step *i-1* as showed by the magnifying glass above a transition in Figure 3.

The SMT formulation is computed recursively: Let A be an automaton (either A_o or A_c), $SMT^{0..i}(A)$ the set of logical constraints related to the automaton A unfolded from step 0 to step i (i.e expressing all execution paths of length i), and TR^{i+1} the set of logical constraints which represents all the possible transitions of A at step $i+1$. The formula below shows the recursive construction of $SMT^{0..i+1}(A)$:

$$SMT^{0..i+1}(A) = TR^{i+1} \wedge SMT^{0..i}(A)$$

SMT Representation of Automata: Combination and Properties Let n and m be the number of execution steps to reach all the observation points of A_o and A_c respectively (with respect to the specific number of iterations in case of loops). To determine if a hardened code is robust, we try to find if a weakness exists. This can be expressed with the following formula:

$$Form \quad = \quad SMT^{0..n}(A_o) \wedge SMT^{0..m}(A_c) \wedge \overline{prop}$$

where *prop* expresses the properties of robustness, as explained in section 3.2.

3.3 Verification and results

The robustness property can be divided into a set of K independent sub-properties denoted $sprop_i$, each focusing on one couple of observation points of A_o and A_c and one variable of $\mathbb{S}$ whose value must be equal for both assembly codes at the observation points. The formula becomes:

$$\begin{aligned} Form \quad &= \quad SMT^{0..n}(A_o) \wedge SMT^{0..m}(A_c) \wedge \overline{\bigwedge_{i=0}^{K} sprop_i} \\ &= \quad \bigvee_{i=0}^{K}(\; SMT^{0..n}(A_o) \wedge SMT^{0..m}(A_c) \wedge \overline{sprop_i} \;) \end{aligned}$$

The formula can thus be divided into a set of sub-formulae, the disjunction of which defines the main formula *Form*. If one sub-formula is satisfiable, then the main formula is satisfiable. Thus, each sub-formula can be solved independently.

We can further simplify the size of the sub-problems to solve by forcing a specific faulty transition in each sub-problem. Thus $Form$ can be expressed as a set of sub-formulae that each focus on a specific fault. Let $A_c^{f_j}$ be the automaton A_c where only the fault transition f_j is represented:

$$Form \quad = \quad \bigvee_{f_j} \bigvee_{i=0}^{K} (\ SMT^{0..n}(A_o) \wedge SMT^{0..m}(A_c^{f_j}) \wedge \overline{sprop_i}\ )$$

All sub-formula can be solved in parallel by different processors, which makes solving the problem significantly faster. Moreover, this technique enables a user to precisely know the instructions, variables of $\mathbb{D}$, and execution steps to be corrupted (this information is given by the faulty transition) in order to get an erroneous value for some variables of $\mathbb{S}$. The corrupted variables of $\mathbb{S}$ at the observation points are also known. This decomposition is therefore helpful for finding and understanding all possible attack paths to bypass the security of an assembly-level countermeasure for a given fault model.

4 Evaluating and Improving Robustness of Countermeasures

This section illustrates the usefulness of our approach by analyzing the robustness of existing protections. We show that our formal approach enables to ensure whether a certain protection is robust with regard to a specific type of fault, and that it can precisely expose any remaining weaknesses.
For each protection, we also propose an improved upon application that offers more extensive security. These improvements were developed with the aid of our formal approach. Note that we do not make any claims concerning the efficiency of these new countermeasures: we can prove that they offer superior protection, but this may come at a high overhead cost. More efficient variants offering the same amount of protection may exist.

We have chosen three existing protections for this evaluation, which are described in detail in the sections to follow.

1. *Memory store verification*, seeing as it is a very basic technique that is often used in industry.
2. *Loop iteration counter duplication*, because it attempts to guard the control flow of a loop, and uses some clever tricks to do so (e.g. inverting the condition of a branch) [20].
3. *instruction duplication*, since the principal idea behind this technique is widely applicable [12].

4.1 Memory Store Verification

A simple and well-known technique to verify whether data has been correctly written to memory, is to load that data from memory into a (free) register immediately after storing it, and comparing it to the register still holding the value that was stored. If both values are the same, then program execution can continue normally. If they differ, this indicates a fault has taken place. In the latter case, any fault handling code or mechanism may be executed to abort or possibly restore execution of the program. Note that, while this protection technique does not guarantee that data cannot be altered in memory after it has been written, it can be combined with error-correcting codes that protect the memory's integrity.

The protection technique described above was implemented (among others) by De Keulenaer et. al. in a prototype link-time code rewriter [20], to show that it is possible to apply those protections automatically, and with an acceptable overhead. The technique we describe here is referred to by the authors as *memory store verification*. Listing 1 shows a sequence of instructions containing a store operation to be protected.

Listing 1. Original code

```
1      mov  r0, #imm
2      mov  r1, @data0
3      subs r0, r0, #1
4      str  r0, [r1]
5      beq  .label
```

Listing 2. Improved protection

```
1      mov  r0, #imm
2      mov  r0, #imm
3      mov  r1 @data0
4      mov  r1 @data0
5      subs rx, r0, #1
6      subs rx, r0, #1
7      str  rx, [r1]
8      str  rx, [r1]
9      beq  .label
10     beq  .label
```

Listing 3. Protected code

```
1      mov  r0, #imm
2      mov  r1, @data0
3      subs r0, r0, #1
4      str  r0, [r1]
5      ldr  r2, [r1]
6      cmp  r0, r2
7      beq  .correct
8      <fault handling code>
9   .correct:
10     cmp  r0, #0
11     beq  .label
```

Listing 3 shows a protected version of the code fragment. The countermeasure aims to protect the store instruction at line 4 in the aforementioned listing against an instruction skip. The instruction at line 10 was inserted to recompute the flags.

Both the original code fragment and the protected variant were transcribed for use by our formal verification framework. This enabled us to find vulnerabilities remaining in the protected code. They relate to the first three instructions, that determine the memory address `r1` and the value of `r0` which should be stored to memory. The wrong value of `r0` can change the outcome of the `cmp` instruction (at line 10 in Listing 3), leading control flow down the wrong path. This cannot be detected by loading the value from memory again and comparing it, since the store operation itself was executed correctly. If the first or the third instruction is skipped, the memory, flags and control-flow may all become corrupted.

Moro et al. proposed a countermeasure pattern [12] which can be applied to protect the store instruction (as well as the instructions that contribute to the value in r0 and r1), and which is based on instruction duplication. Note that this scheme offers fault resilience, whereas the protection applied by De Keulenaer et al. offers fault detection. Listing 2 shows the result of applying the aforementioned countermeasure pattern proposed by Moro et al., and further optimizing it. Using our framework, we were able to prove its robustness against a single instruction skip fault. Besides protecting the store operation itself, which was already accomplished in the application of the original protection, this protection ensures that the correct value will always be written to the correct memory address. The direction of control flow at the end of the fragment is also protected.

4.2 Loop Iteration Counter Duplication

Another countermeasure implemented by De Keulenaer et al. aims to protect the number of iterations of a loop [20]. An unprotected code fragment is shown in Listing 4, whereas Listing 6 shows the same code fragment after protection.

Listing 4. Original code

```
.preheader:
    mov  r2, r0
.body:
    add  r2, r2, #1
    sub  r1, r1, #1
    cmp  r1, #0
    bne  .body
.after:
```

Listing 5. Improved protection

```
    mov  r2, r0
    mov  r5, r1
.body:
    add  r2, r2, #1
    sub  r1, r1, #1
    sub  r5, r5, #1
    cmp  r1, r5
    bne  .fault
    cmp  r1, #0
    cmp  r1, #0
    bne  .body
    bne  .body
.end:
    ...
.fault:
    <fault handling code>
```

Listing 6. Protected code

```
.preheader:
    mov  r2, r0
    mov  r5, r1
.body:
    add  r2, r2, #1
    sub  r1, r1, #1
    sub  r5, r5, #1
    cmp  r1, #0
    bne  .check2
.check1:
    cmp  r5, #0
    beq  .after
    <fault handling code>
.check2:
    cmp  r5, #0
    bne  .body
    <fault handling code>
.after:
```

The loop iteration counter duplication protection was proven secure by our formal verification framework; i.e. it is able to protect the number of iterations of the loop by successfully detecting any single instruction skip, as claimed by the authors.

However, this protection does have one inherent weakness, because the counter is checked against its duplicate only when exiting from the loop. If, because of a fault, either the iteration counter or its duplicate is not decremented during a certain iteration, execution may continue for a long time before this is detected (e.g. for a high value of the counter). While it is practically unfeasible for an attacker to prevent both the iteration counter and its duplicate from being decremented during a single loop iteration, he may be able to prevent the iteration counter from being decremented during one iteration, and subsequently prevent the duplicate counter from being decremented many iterations later, before either counter hits zero. If he succeeds in doing this, he has successfully changed the number of loop iterations without this being detected. Indeed, such an attack does imply that two faults are injected. However, given the possibility of a large enough time-frame, we deem this to be feasible.

For this specific application of the loop iteration counter duplication, a possible improvement is shown in Listing 5. In this variant, the original loop counter is compared to its duplicate during every iteration of the loop. This means any modification to either the counter or its duplicate will never go undetected for more than one iteration of the loop. By duplicating the `cmp` and `bne` instructions, we ensure that the loop will always be executed the right amount of times. This is because it is practically unfeasible for an attacker to inject a fault in two subsequent instructions.

4.3 Instruction Duplication

Barenghi et. al. explored countermeasures and detections against fault attacks based on software redundancy [10]. Listings 7 and 8 illustrate how these techniques can be used to protect a load instruction.

Listing 7. Original code

```
    ldr  r4, [r7]
```

Listing 8. Protected code

```
    ldr  r4, [r7]
    ldr  r12, [r7]
    cmp  r4, r12
    bne  .fault
    ...
.fault:
    <fault handling code>
```

Listing 9. Improved protection

```
    mov  r7, @data0
    ldr  r4, [r7]
    ldr  r12, [r7]
    mov  r7, @data0
    ldr  r12, [r7]
    msr  apsr, #0
    cmp  r4, r12
    beq  .end
    b    .fault
.end:
    ...
.fault:
    <fault handling code>
```

Using our framework, we were able to prove the robustness of this protection technique against a single instruction skip fault, as claimed (but not proven) by the authors. We also determined whether this protection technique is robust with regard to the SIR fault model. The results are shown in Table 1.

The first column in this table shows the instruction which is modified by a simplified instruction replacement fault. For that instruction, the second column shows which registers are alive after the instruction has been executed, and the third column shows how a replacement can lead to a faulty value of one of those registers. Note that not all values of x lead to a faulty value of one of the live registers. At this time, our tool does not yet output the complete set of values for x that can corrupt one of the registers. While we have planned this as future work, for this manuscript we have deduced the success conditions given in the last column from the output of our tool.

instruction replaced	**alive variables**	**new instruction**	**success condition**
`ldr r4, [r7]`	`r4`, `r7`	$\texttt{r7} \leftarrow x$	$\texttt{Mem[x]} = \texttt{r4} \land \texttt{r4} \neq \texttt{Mem[r7]}$
`ldr r12, [r7]`	`r4`, `r12`	$\texttt{r4} \leftarrow x$	$x = \texttt{r12} \land \texttt{r12} \neq \texttt{Mem[r7]}$
`cmp r4, r12`	`r4`, `flags.Z`	$\texttt{r4} \leftarrow x$	$x \neq \texttt{Mem[r7]} \land \texttt{flags.Z} = 1$
`bne .fault`	`r4`	$\texttt{r4} \leftarrow x$	$x \neq \texttt{Mem[r7]}$

Table 1. Results of the robustness evaluation: attack paths to corrupt `r4`

For a more detailed discussion, let's focus on the first row in the table. Registers `r4` and `r7` are alive after the execution of `ldr r4, [r7]`. Among them, only the corruption of register `r7` by an address x, the contents of which must differ from memory case `Mem[r7]`, can affect the final state of the register `r4`.

Table 1 shows that a simplified instruction replacement fault can bypass the security of any instruction in the protected code fragment. However, not all of these faults have the same probability of occurrence. Only one specific value of x can lead to an erroneous value of `r4` if `ldr r12, [r7]` is attacked (if x is equal to the initial value of `r12`), whereas only one specific value of x will not corrupt the final state of `r4` if `bne .fault` is attacked and `r4` corrupted.

The insights we have acquired allowed us to come up with a more robust application of the instruction duplication countermeasure, one that can be used to protect the load instruction against a simplified instruction replacement fault. The improved upon application is shown in Listing 9. In this listing, the value to be loaded from memory is stored at address `@data0`. As we have shown earlier, register `r7`, which contains this address, is a sensitive register. In order to prevent `r7` from getting an erroneous value which, in turn, leads `r12` to have the wrong value loaded from memory, the latter had to be re-affected by `@data0` (at line 4 in the aforementioned listing), between the first and second occurrence of `ldr r4, [r7]`. That instruction was in fact duplicated to prevent a single fault on this instruction to corrupt `r4` with the initial value of `r12`: a fault affecting the first `ldr r4, [r7]` will not bypass the countermeasure since the second one will affect `r12` with the correct value. The countermeasure is thus able to detect an erroneous value in r4. A fault affecting the second ldr r4, [r7] will always be detected as well, since either r4 or r12 already have the correct value.

In Listing 8, cmp r4, r12 can lead to corruption of r4 if the replacing instruction affects r4 and if the Z (zero) flag is set to true. Keeping the value true in flags.Z prevents the branch instruction from jumping to .fault. To avoid this, msr apsr, #0 was added at line 6 in Listing 9. This instruction sets all flags to false, so the branch instruction will always go to the fault detection state if r4 is corrupted.

The fourth instruction of Listing 8 is the most vulnerable one. If r4 is corrupted, only one value for x doesn't lead to an erroneous state of r4. Also, the last instruction of an application of a countermeasure is always more difficult to secure, since there is no instruction after its execution to detect the fault. One solution is to execute the last instruction (whose purpose is to branch to the fault detection state) only if a fault occurs before its execution. Otherwise, the countermeasure must have already detected that r4 has the correct value. This mechanism is implemented at lines 8 and 9 of Listing 9. This technique ensures that b .fault is executed only if a fault has already occurred. It also ensures that the control flow is subverted away from the countermeasure's basic block before it reaches the last instruction; the code that starts at label end can detect the fault.

Hence, the improved countermeasure shown in Listing 9 is robust against the simplified instruction replacement fault model. It guarantees that register r4 (and, by extension, r12) will always have the corrected value loaded from the memory address r7. Developing an improved countermeasure so quickly would have been very difficult, or even impossible, without the aid of our formal verification framework, because of the human brain's limited capacity to enumerate all vulnerabilities that are present in a fragment of assembly code.

4.4 Discussion

Seeing as the proof is divided in a set of independent sub-proofs (see Section 3.3), the total sequential verification time is the sum of the verification times of each sub-formula. The verification time of a sub-formula depends on the property being proved. Alos, for both the sub-proof and the proof in its entirety, the verification time is sensitive with regard to the complexity of the assembly code (register dependency, control flow, instruction mix, ...), its length and the fault model taken into consideration. Explaining why, however, would lead us too far away from the context of this paper.

Concerning the time needed for our approach for the robustness evaluation, proving the robustness of the improved countermeasure (Listing 9) required the longest time, which was 10.7 s using a laptop computer (Intel® Core$^{\text{TM}}$ i3-3120M, CPU 4 × 2.50GHz). In this case, the proof was divided into about 100 sub-proofs. The longest verification time for one such sub-proof was 90 ms. Without this decomposition, not only would the evaluation time have been at least one order of magnitude higher, but it would not have been possible to easily obtain the different attack paths in the initial version, and to construct the improved countermeasure.

For analyzing an entire code, our approach requires to decompose this code into pieces, the robustness of each piece to be analyzed separately. With the fault models considered in this paper and the robustness defined as the equivalence of the content of live registers and memory at some control points, this decomposition into small sequences of instructions enables the verification of large code.

5 Conclusions

We presented a formal verification method for evaluating the robustness of existing fault-injection countermeasures, which models code fragments as simple automata, and implemented this functionality in the form of a tool. This tool can also assist in improving existing countermeasures, as well as in the development of new countermeasures, as we have shown in this paper. Currently, the ARMv7-M (Thumb2) ISA is supported, but we estimate the porting of our tool to another assembly instruction set to be a matter of weeks. We believe the features of our formal verification framework are promising, and that there are many possible directions for this research as far as future work is concerned.

What we would like to do next, is to investigate metrics of robustness to classify countermeasures according to their strength (i.e. how robust they are against a certain type of fault). We also plan to study possible ways to reduce the verification cost for supporting a full instruction replacement fault model (which takes into account jumps and store instructions). Another research direction we are currently exploring is how we can combine the functionality of the formal verification framework with an automated code rewriting tool. This would allow to generate an automaton description representation of every single instance of an applied protection, which in turn will allow to verify whether all of the sensitive code that is to be protected, is in fact robust against a certain type of fault.

References

1. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE **94**(2) (2006) 370–382
2. Bhasin, S., Maistri, P., Regazzoni, F.: Malicious wave: A survey on actively tampering using electromagnetic glitch. In: International Symposium on Electromagnetic Compatibility. (2014) 318–321
3. Boneh, D., DeMillo, R., Lipton, R.: On the Importance of Checking Cryptographic Protocols for Faults. In: Advances in Cryptology — EUROCRYPT. Volume 1233 of LNCS. (1997) 37–51
4. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: 17th International Cryptology Conference on Advances in Cryptology. CRYPTO (1997) 513–525

5. Amiel, F., Villegas, K., Feix, B., Marcel, L.: Passive and Active Combined Attacks: Combining Fault Attacks and Side Channel Analysis. In: Workshop on Fault Diagnosis and Tolerance in Cryptography FDTC. (2007) 92–102
6. Rauzy, P., Guilley, S.: A Formal Proof of Countermeasures Against Fault Injection Attacks on CRT-RSA. Journal of Cryptographic Engineering JCEN **4**(3) (2014) 173–185
7. Lalande, J.F., Heydemann, K., Berthomé, P.: Software Countermeasures for Control Flow Integrity of Smart Card C Codes. In: Computer Security - ESORICS. Volume 8713 of LNCS. (2014) 200–218
8. Asghari, S., Abdi, A., Taheri, H., Pedram, H., Pourmozaffari, S.: SEDSR: soft error detection using software redundancy. Journal of Software Engineering and Applications **5** (2012) 664
9. Goloubeva, O., Rebaudengo, M., Reorda, M., Violante, M.: Improved software-based processor control-flow errors detection technique. In: Reliability and Maintainability Symposium. (2005) 583–589
10. Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., Regazzoni, F.: Countermeasures Against Fault Attacks on Software Implemented AES: Effectiveness and Cost. In: 5th Workshop on Embedded Systems Security, ACM (2010) 7:1–7:10
11. Reis, G., Chang, J., Vachharajani, N., Rangan, R., August, D.: SWIFT: Software Implemented Fault Tolerance. In: International Symposium on Code Generation and Optimization. (2005) 243–254
12. Moro, N., Heydemann, K., Encrenaz, E., Robisson, B.: Formal verification of a software countermeasure against instruction skip attacks. Journal of Cryptographic Engineering **4**(3) (2014) 145–156
13. Verbauwhede, I., Karaklajic, D., Schmidt, J.: The fault attack jungle-A classification model to guide you. In: Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), IEEE (2011) 3–8
14. Goloubeva, O., Rebaudengo, M., Reorda, M., Violante, M.: Soft-error detection using control flow assertions. In: 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems. (2003) 581–588
15. Bayrak, A., Regazzoni, F., Novo, D., Ienne, P.: Sleuth: Automated Verification of Software Power Analysis Countermeasures. In: Cryptographic Hardware and Embedded Systems (CHES). Volume 8086 of LNCS. (2013) 293–310
16. Eldib, H., Wang, C.: Synthesis of Masking Countermeasures against Side Channel Attacks. In: 26th International Conference on Computer Aided Verification. (2014) 114–130
17. Potet, M.L., Mounier, L., Puys, M., Dureuil, L.: Lazart: a symbolic approach for evaluation the robustness of secured codes against control flow fault injection. In: ICST. (2014)
18. Chetali, B., Nguyen, Q.H.: Industrial Use of Formal Methods for a High-Level Security Evaluation. In: Formal Methods. Volume 5014 of LNCS. (2008) 198–213
19. Moro, N.: Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués. PhD thesis, UPMC, France (2014)
20. De Keulenaer, R., Maebe, J., De Bosschere, K., De Sutter, B.: Link-time smart card code hardening. International Journal of Information Security (2015) 1–20