A tool assisted methodology to harden programs against multi-faults injections

Etienne Boespflug, Abderrahmane Bouguern, Laurent Mounier, and Marie-Laure Potet

VERIMAG
University of Grenoble Alpes (UGA), Grenoble, France
Firstname.Lastname@univ-grenoble-alpes.fr

Abstract. Fault attacks consist in changing the program behavior by injecting faults at run-time in order to break some expected security properties. Applications are hardened against fault attack adding countermeasures. According to the state of the art, applications must now be protected against *multi-fault injection* [1,2]. As a consequence developing applications which are robust becomes a very challenging task, in particular because countermeasures can be also the target of attacks [3,4]. The aim of this paper is to propose an assisted methodology for developers allowing to harden an application against multi-fault attacks, addressing several aspects: how to identify which parts of the code should be protected and how to choose the most appropriate countermeasures, making the application more robust and avoiding useless runtime checks?

Keywords: multiple fault-injection; code analysis; software countermeasure; dynamic-symbolic execution.

1 Introduction

Fault injection is a powerful attack vector, allowing to modify the code and/or data of a software, going much beyond more traditional intruder models relying "only" on code vulnerabilities and/or existing side channels to break some expected security properties. This technique initially targets security critical embedded systems, using physical disturbances (e.g., laser rays, or electro-magnetic fields) to inject faults. However, it may now also concern much larger software classes when considering recent hardware weaknesses like the so-called Rowhammer attack [5,6], or by exploiting some weaknesses in the power management modules [7,8,9]. Furthermore, in the growing domain of IoT, security is based on very sensitive operations such as boot-loading or Over-the-Air firmware update which must be protected against fault injections[10,11].

As a result, programs must be hardened against fault injection, combining hardware and/or software *countermeasures* aiming to detect runtime security violations. According to the state of the art, applications must now be protected against (spacial or temporal) *multi-fault injection* [1,2], namely when several faults can be injected at various times or locations during an execution. As a

consequence developing applications which are robust becomes a very challenging task and a cat-and-mouse game, in particular because countermeasures can be also the target of attacks [3,4]. The aim of this paper is to propose an assisted methodology for developers allowing to harden an application against multi-fault attacks, addressing several aspects: how to identify which parts of the code should be protected and how to choose the most appropriate protection schemes, making the application more robust and avoiding useless runtime checks?

There exist tools adding countermeasures, generally at compilation-time. They are dedicated to particular fault models (data modification, instruction skip, flow integrity) [12,13,14], for instance by adding redundant checks or duplicating idempotent instructions. Nevertheless these tools target single fault robustness, where countermeasures themselves cannot be faulted. Furthermore such countermeasures are added in a brute force approach, based on (coarse-grained) directives given by the developers, indicating which parts of the code must be protected. Such an approach is no longer realistic when multi-faults must be taken into account, since more complex countermeasures must be considered, potentially adding unnecessary performance/size costs.

The objective of this paper is to address the issue of assisting security developers in the countermeasure insertion process. In particular, we provide the following contributions:

1. we formulate the problem of robustness comparison in presence of multi-faults;
2. we propose a methodology to analyze countermeasures "in isolation", both in terms of classes of attacks they detect and considering their own attack surface;
3. we propose an algorithm allowing to harden applications starting from identified vulnerabilities, depending on countermeasure properties;
4. we provide an implementation of this approach, based on the Lazart tool [15] and evaluate our approach on a benchmark of code examples.

Section 2 introduces multiple fault-injection and countermeasures through a motivating example and presents the Lazart tool. Section 3 proposes definitions for robustness evaluation and comparison dedicated to multi-faults. Section 4 describes the proposed methodology for analyzing countermeasures in term of their adequacy and weakness against established fault models. Section 5 presents our countermeasure placement algorithms and illustrates them on several examples. Finally, Section 6 discuss related works, limitations of our solution and future directions.

2 Fault injection attacks

In this section we introduce fault injection attacks through a simple example and the tool Lazart which is used to analyze robustness against fault attacks. Then we propose a hardened version of our example illustrating classical countermeasures and introduce our contributions.

2.1 A fragile byteArrayCompare version

Listing 1.1 is an excerpt of a `byteArrayCompare` function used in the verifyPIN collection taken from the FISSC public benchmark[16]. Such a function compares two PIN codes and returns true if they are equal. `BOOL_TRUE` and `BOOL_FALSE` are robust encoding of boolean constants true and false[1]. The security property we want to guarantee is that `result` is true if only if the two PIN codes are equals (`a1[0..size-1]==a2[0..size-1]`.

```
1 BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size){
2       int i;
3       int result = BOOL_TRUE;
4       for(i = 0; i < size; i++)
5             if(a1[i] != a2[i]) result = BOOL_FALSE;
6       return result; }
```

Listing 1.1. Fragile byteArrayCompare function

This example is insecure against fault injection consisting in inverting control flow condition. To show that we use Lazart [15], a tool analyzing the robustness of a software under multi-fault injections. It takes as input a C code, a fault model, a security property and exhaustively find all attack paths. Lazart relies on a 2-steps approach:

- First, a *high order mutant* is generated from the LLVM representation of the program. This mutant statically encodes all the possible injected faults (as symbolic boolean values) according to a chosen fault model
- Then, a *dynamic-symbolic exploration*, performed by Klee [17], produces a representative path for all the successful attacks violating the security property.

Various fault models are supported by Lazart targeting control flow or data modifications (as illustrated in Section 4). Table 1 line V1 gives the results supplied by Lazart for this first version when `size` is fixed to 4 assuming all bytes of the two input arrays `a1` and `a2` are different. We consider here a limit of 8 faults. All these attacks violate the security property.

Table 1. Lazart results for the `byteArrayCompare` versions

Order	0 fault	1 fault	2 faults	3 faults	4 faults	5 faults	6 faults	7 faults	8 faults
V1	0	**1**	**1**	**1**	**2**	0	0	0	0
V2	0	0	**1**	0	**1**	0	**1**	0	**2**

The 1-fault attack consists in inverting the loop condition $i < size$ (line 4 of Listing 1.1), stopping directly the array comparison. The 2-faults, 3-faults and one of the 4-faults attack consist in inverting the test of line 5, one, two or three times respectively, and then stopping the loop. The last 4-faults attack consists in inverting four times the test of line 5 and exiting the loop normally.

[1] Classically `0xAA` and `0x55` to be resistant to bit flip.

2.2 A robust byteArrayCompare version

We propose in Listing 1.2 a more robust version of the function byteArrayCompare, adding classical redundant-check countermeasures: 1) a test line 10 checking the exit value of the loop and 2) a systematic verification of the result of condition (lines 6 and 8). Calls to the function `atk_detected` stop the execution, signaling the detection of an abnormal behavior.

```
1  BOOL protected_byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size){
2        int i;
3        int result = BOOL_TRUE;
4        for(i = 0; i < size; i++) {
5          if(a1[i] != a2[i]) {
6             if(a1[i] == a2[i]) atk_detected();
7             result = BOOL_FALSE;}
8          else if(a1[i] != a2[i]) atk_detected(); }
9        if(i != size)  atk_detected();
10       return result; }
```

Listing 1.2. Example: Secure byteArrayCompare function

In a multi-fault context, the attacker is able to bypass countermeasures. For instance here she can invert the extra tests (lines 6, 8 and 9). Results supplied by Lazart for this new version (still for the test inversion fault model) are given on Table 1, line V2. Therefore, the 1-fault attacks found in the fragile function is no longer possible and it now requires two test inversions: the loop condition and its associated check line 9. More generally each previous attack now requires to double the number of injected faults. As we can see, added countermeasures make our function more robust, but unfortunately they also come with their own attack surface (here our encoding of countermeasures is sensitive to test inversion).

2.3 Attack information supplied by Lazart

Attacks will be represented by sequences of fault occurrences consisting in pairs (injection point, fault model). For instance the first attack of Table 1 is represented by $<$ IP4(TI), IP9(TI)$>$, where IP4 is the fault injection point of line 4 and TI denotes "test inversion". From that we can compute how many times a fault injection point occurs into successful attacks. Table 2 shows results associated to Table 1. In particular we can observe that attacking IP6, corresponding to an added countermeasure, never results in a successful attack. As a consequence this countermeasure can be removed, without impacting the robustness of our implementation (for the considered security property). In the same way if we only consider order 2 attacks, countermeasures lines 5 and 8 are no longer useful. Other configurations where some countermeasures can be safely removed can be found in [18].

In the context of multi-faults, some attacks can be considered as *redundant*. An attack b will be consider as redundant with respect to an attack a iff a is a *proper prefix* of b, denoted by $a \leq b$, $\leq$ being a partial order relation.

vspace*-1em

Table 2. Lazart Hotspots analysis for Listing 1.2

IP line	0 fault	1 fault	2 faults	3 faults	4 faults	5 faults	6 faults	7 faults	8 faults	Total
line 4	0	0	1	0	1	0	1	0	1	**4**
line 5	0	0	0	0	1	0	2	0	7	**10**
line 6	0	0	0	0	0	0	0	0	0	**0**
line 8	0	0	0	0	1	0	2	0	7	**10**
line 9	0	0	1	0	1	0	1	0	1	**4**

Definition 1. *Minimal attacks*
Let E be a set of attacks. $Minimal(E)$ is the smallest subset of E such that every attack b in E is represented by a unique attack a in $Min(E)$ such that $a \leq b$.

In order to give a more synthetic view Lazart can compute the subset of minimal attacks and how many attacks they represent. For the example of Listing 1.1 no attack is redundant, but considering at first minimal attacks happens to be useful for larger examples [19]. Lazart supplies also others results allowing to evaluate the coverage of the analysis, such as the number of explored paths, if some timeout has been raised, etc. These information are useful to assess the completeness of the analysis.

2.4 Our objectives and contributions

In the context of single fault, hardening a code generally follows a "try and test" approach: countermeasures are added and robustness is checked through a new analysis (as such a secure implementation is now expected to become robust against single-fault attacks). Nevertheless this process is not worth considering in a multi-fault context since it would introduce a very detrimental overhead. The aim of this paper is to propose several placement algorithms allowing to add countermeasures advisedly targeting to avoid unnecessary code. To do that we propose a methodology for stating and analyzing countermeasures "in isolation" in order to formally establish properties such as the *adequacy* of a countermeasure w.r.t. a given fault model and its inherent *attack surface*. Countermeasures we target are systematic countermeasures detecting attacks, like test or load duplication, and, more generally, systematic countermeasures allowing to detect errors in data or control flows.

To the best of our knowledge, this approach is innovative. As pointed out before, generally tools add countermeasures in a systematic way (e.g., *stack canaries* are added by compilers to any functions satisfying some basic syntactic criteria) without precisely taking into account which control locations should be protected w.r.t. the security properties which have to be enforced. This situation introduces potentially serious and unnecessary overheads. On the other hand, methods has been proposed to prove hardened programs [20,21,4], or to verify the efficiency of a given form of countermeasure against attacker models [22,23]. All these approaches are developed in the context of single fault and adapted

to a particular countermeasure or fault models. On the contrary, the approach we consider here is general, modular, able to cope with multi-fault attacks and it addresses the problem of optimization of countermeasures placements. As a first step we propose in the next section a framework to properly compare the robustness of several versions of a same code in a multi-faults context.

3 Robustness metrics for multi-faults

In the context of certification for high levels of security [24] applications are submitted to vulnerability analyses (the AVA class of Common Criteria), conducted by experts team (as ITSEF laboratory). For instance rating physical attacks for smart cards or similar devices is established by the JIL Hardware Attacks Subgroup [25] on the based on a set of relevant factors: elapsed time, expertise, knowledge of the target of evaluation, access to it, ...

At the level of code-based simulation tools some metrics are used such as the number of successful attacks, potentially weighted by the total number of attack or the attack surface [26]. Nevertheless these metrics are not really used in practice to compare robustness of applications: the try and test approach turns out to be suffisant when single fault is considered. On the contrary, in the context of multi-faults we have to formalize how implementations can be compared, that we propose here. Furthermore we will use our formalization to validate our methodology and algorithms showing that we improve the robustness of applications in adding countermeasures.

3.1 Definition of multi-fault robustness

Generally simulation tools give the number of attacks for a given *golden run* starting from a fixed initial state. A simulated trace t is a finite sequence of transitions corresponding to a nominal execution steps or faulted steps, starting from an initial state. In the following $init(t)$ denotes the initial state of t and $fault(t)$ the number of faulted transitions into t. For a success condition S (generally the negation of a security property), a set of fault models m, simulated traces T of a program P can be partitioned into the following way:

- $T_N(m, S)$: traces obtained under the nominal execution (without fault)
- $T_D(m, S)$: faulted traces that are detected by a countermeasure
- $T_S(m, S)$: successful attacks (verifying S and not detected)
- $T_F(m, S)$: non detected faulted traces that do not verify S

Definition 2. *Input vulnerability characterization*

Let P be a program, i an initial state (including inputs), m a set of fault models and S a successful condition. We define as $Vuln_\mu(P, i, m, S)$ the attack function associated to the input i defined from the rank 0 to μ:

$$Vuln_\mu(P, i, m, S) \hat{=} f \in 0..\mu \to \mathbb{N} \mid \forall n \in 0..\mu \ (f(n) = \#\{t \in T_S(m, S) \mid init(t) = i \ \wedge \ fault(t) = n\}$$

Definition 2 can be extended to a set of initial states I. For instance Table 1 describes $Vuln_\mu(P, I, m, S)$ with $\mu = 8$, $I = \{a2[0..3], a1[0..3], 4) \mid \forall i \in 0..3\ a1[i] \neq a2[i]\}$, $m = \{Test\_inversion\}$ and S be `result==BOOL_TRUE`.

Definition 3. *Robustness level*

If I is the set of all initial states and $Vuln_\mu(P, i, m, S)(i) = 0$ for all $i \in 1..\mu$ and for all $i \in I$ we say that P is robust up to μ faults.

3.2 Robustness comparison

Here we target to compare a program P with an hardened version of P. For instance we want to state that the secure version of function `byteArrayCompare` is more secure than the initial version, up to order 8.

Definition 4. *Robustness comparison*

Let P be a program, I a set of initial states, μ the order to consider, m the set of fault models and S the successful condition. Let P' be an hardened version of P. Robustness comparison is defined as follow:

$$P' \geq^{rob}_{\mu,I,m,S} P \ \hat{=}$$
$$\forall i \in I\ \forall n \in 1..\mu \sum_{j=1}^{n} Vuln_\mu(P', i, m, S)(j) \leq \sum_{j=1}^{n} Vuln_\mu(P, i, m, S)(j)$$

Our definition does not only compare the total number of attacks for a given order but all intermediary sums. Thanks to this definition, several nice properties hold as *monotonicity* ($P' \geq^{rob}_{n+1,I,m,S} P \Rightarrow P' \geq^{rob}_{n,I,m,S} P$) and transitivity ($P'' \geq^{rob}_{\mu,I,m,S} P' \wedge P' \geq^{rob}_{\mu,I,m,S} P \Rightarrow P'' \geq^{rob}_{\mu,I,m,S} P$).

3.3 Discussion

A classical approach to evaluate the robustness of a code against fault attack is to count the number of faults issued from a given golden run, where both the inputs and the initial states are fixed in advance. Several works [27,28,16] use *metrics* to compare protected program versions in this setting. The robustness comparison relation we propose can be seen as a generalization of these metrics, taking into account a set of initial states as well as multi-faults and still offering good properties. Note that this comparison relation is relevant only for two programs having the same nominal behaviors (i.e. the same functionality without fault). Moreover, it is a partial relation. Typically, two programs P and P' can become incomparable up to $n+1$ faults if P' is more robust than P up to n faults and, for $n+1$ faults, the sum of the attacks of program P' is greater than the one of P. Finally, thanks to our symbolic execution based approach, computing this relation needs to consider in practice only one representative attack per execution path (even potentially reduced to *minimal attacks* as defined in definition 1) hence allowing to reduce metrics explosion due to the added countermeasures (according to the dilution paradox [26]).

4 Countermeasures analysis

In this section, we propose a systematic approach for analyzing countermeasures " in isolation " in terms of classes of attacks they detect. Combined with their proper attack surface (section 5.1) we are able to construct what we called the "countermeasure catalog", used later to choose adapted countermeasures for hardening implementations. Our approach is based on the encoding of several generic schemes, described by C codes, associated with their pre and postconditions. We use the weakest-precondition WP plugin of Frama-C and the ACSL annotation language [29].

1. **Mutation schemes** will characterize a fault model as how a fault impacts the behavior of sensible instructions. Postcondition states the nominal behavior of the instruction as well as the behavior is impacted by a fault.
2. **Countermeasure schemes** will describe how abnormal behavior will be detected. Postcondition characterized behaviors that are not blocked.
3. **Protected schemes** will describe how countermeasures are inserted w.r.t. mutation schemes. The expected postcondition is the nominal behavior associated to the mutation scheme (all faults provoking a variation of the nominal behavior must be blocked).

We will illustrate our approach on three fault models: test inversion, data load modification (when we read a memory value) and Else-following-Then. These fault models are very classical ones, in particular the Else-following-Then model describes when the jump at the end of the then block is skipped (or transformed into a nop operation). We also consider three countermeasures: test duplication, load duplication and a lightweight control flow integrity based on block signature.

4.1 Mutation schemes

We consider a fault model as a set of mutation schemes describing how the code is transformed under a fault injection. The `fault` parameter represent the presence of a fault (0 if no fault, different from 0 otherwise). The postcondition describes the expected nominal behavior (when `fault` equals 0) as well as the result of a fault. Thanks to the ACSL language we can clearly distinguish nominal and faulty behaviors. Listing 1.3 describes the mutation scheme for test inversion, when inverting the result of a conditional jump. Other fault models are given in Appendix A.

```
/*@ assigns \nothing ;
  @ behavior nominal :
  @    assumes  fault==0 ;
  @    ensures  C!=0 ==>  \result==br_then ;
  @    ensures  C==0 ==>  \result==br_else ;
  @ behavior faulted :
  @    assumes  fault!=0 ;
  @    ensures  C!=0 ==>  \result==br_else ;
  @    ensures  C==0 ==>  \result==br_then ; @*/
int mutation_ti(int C, int br_then, int br_else, int fault)
   { if((C && !fault) || (!C && fault)){ return br_then; }
```

```
12      else { return br_else; }  }
```

Listing 1.3. Test inversion mutation scheme

All fault schemes (listings 1.3 and 1.7, 1.8 in Appendix A) are proved by the plugin Frama-C/WP.

4.2 Countermeasure schemes

Countermeasures calls a detection function (atk_detected) which stops the execution. Listing 1.4 describes the detection function and a countermeasure duplicating the condition. Listing 1.5 describes a more sophisticated countermeasure, with a proper variable representing which block must be reached. Function `sign` records the name of the next block and function `check` verifies that the reached block corresponds to the expected one.

```
1 /*@  ensures \false;
2   @ assigns \nothing ;@*/
3 void atk_detected() { exit(0); }
4
5 /*@ ensures   C!=0;
6   @ assigns \nothing ; @*/
7 void TestDup (int C) { if(!C) atk_detected(); }
```

Listing 1.4. Test duplication countermeasure

```
1 int RTS;
2
3 /*@ ensures  ((C!=0 ==>  RTS==id_true)) ;
4   @ ensures  ((C==0 ==>  RTS==id_false)) ;
5   @ assigns RTS; @*/
6 void sign(int C, int id_true, int id_false)
7 { if(C) RTS = id_true; else  RTS = id_false; }
8
9 /*@ ensures (RTS==id) ;
10  @ assigns \nothing; @*/
11 void check(int id) { if(RTS!=id) atk_detected(); }
```

Listing 1.5. Signature based countermeasure

Function `atk_detected` never normally terminated. Each countermeasure targets to block an erroneous control flow. The postcondition describes the final state ensured by the countermeasure. All these functions are proved by the WP/Frama-C plugin, as well as the load duplication countermeasure given in Appendix A.

4.3 Protected schemes

The last step consists to prove the adequacy of a countermeasure against a fault model. To do that we combine countermeasures with a mutation scheme targeting to block abnormal behaviors introduced by the mutation. Listing 1.6 shows how a condition can be protected by test duplication against test inversion. The postcondition is build in a systematic way stating that we terminate normally with the nominal behavior of the mutation scheme (the nominal behavior of Listing 1.3 for our example below). That means that no fault occurs or a faults occurs but without impact on the nominal behavior.

```
1 /*@ behavior nominal :
2   @    ensures  ((C!=0 ==>  \result==br_then)) ;
3   @    ensures  ((C==0 ==>  \result==br_else)) ;
4   @    assigns \nothing ; @*/
5
6 int protected_TI_DT(int C, int br_then, int br_else, int fault){
7     if((C && !fault) || (!C && fault))
8       { TestDup(!C); return br_then; }
9     else { TestDup(C); return br_else; } }
```

Listing 1.6. The insertion of the TD CM in the TI mutation scheme

Listings in Appendix A section A.3 respectively give protected versions for the load modification protected by load duplication (listing 1.9) and for the test inversion and the Else-following-Then fault models protected by the Signature countermeasure (listings 1.11 and 1.12). All these protected versions (including listing 1.6) are proved by the WP/Frama-C plugin.

Contrary to [4] we do not prove the reduced postcondition `fault==0` but we impose to preserve the nominal behavior of the instruction impacted by a fault. Our approach is more general: for instance we admit a fault that does not impact the nominal behavior, as an attack by test inversion when the two targeted blocks are identical or the data load modification when we inject the right value. On the contrary, for listing 1.13, we can strengthen the postcondition with the clause `ensures fault==0`.

4.4 Results and discussions

Proposition 1. *Adequate countermeasure.*

Let m be a fault model characterized by a mutation scheme MS with a nominal behavior NB. A protected scheme PS (embedding a countermeasure) is adequate for m if and only if PS verifies the postcondition NB. Then, according to definition 3 we can state that PS is robust up to order 1, for the fault model m, for all I verifying the precondition of PS and for the success condition $\neg NB$.

Table 3. Adequacy of countermeasures against fault models

Fault model \ Countermeasure	Test duplication	Load duplication	Block signature
Test inversion	adequate	-	adequate
Else following Then	KO	-	adequate
Data load modification	-	adequate	-

Table 3 describes the result of proofs of protected programs we developed. They are all proved by the WP/FRama-C plugin (adapted) except Listing 1.13 that corresponds to an erroneous countermeasure (KO). An empty case means that it does not make sense to consider this combination of fault model and countermeasure. Due to the modular proofs in WP/Frama-C we can replace a sensitive instruction (as a test, a load or and then-else structure) by one of its protected versions without modifying its nominal behavior. For instance we can

replace the instruction `if(C) return br_then; else return br_else;` by `if(C){TestDup (!C); return br_then;} else {TestDup(C); return br_else;}`.

To conclude, the notion of adequacy introduced in Proposition 1 tells us whether a countermeasure protects a sensitive code pattern against a single-fault attack model. The proposed methodology is based on the preservation of the nominal behavior, that can be carefully specified (the semantics of the code pattern as well as the `assigns` clause that allows to embed our protected scheme in a larger code without unpredictable side effects). On the contrary the impact of a fault in mutation scheme is not really useful here but can be exploited later if we want to establish some properties in the presence of faults.

5 Our methodology for hardening program

Starting from a set of attacks, the aim is to propose a methodology for assisting countermeasures placement in the more appropriate way. In the context of multi-faults, countermeasures come with their proper attack surface (the code which is added), and fault injections on this code can produce new execution traces, potentially producing new successful attacks. Because our objective is to propose an alternative to the unrealistic brute force "try and test" approach when multi-faults is considered we want to study the weakness of countermeasures in a modular way. The next section explains how this analysis can be conducted.

5.1 Countermeasure protection level

We define (and compute) the **protection level** of a countermeasure, which characterizes the minimal number of faults allowing to bypass a countermeasure. Bypassing a countermeasure means the countermeasure terminates normally (without call to the function `atk_detect`) but violating its postcondition (i.e. states that must be normally blocked).

Definition 5. *Protection level*
Let CM be a countermeasure scheme defined by its postcondition R and M a set of fault models. Let $pl(CM, M) = l$ the level of protection of CM w.r.t. M defined as:
Condition 1. *l is the minimal number of necessary faults producing an attack violating R: $l = min\{(fault(t) \mid t \in T_S(M, \neg R)\}$ (or ∞ if no successful attack exists).*
Condition 2. *any traces t such that 0<fault(t)<l are blocked : $\forall t\ (0 < fault(t) < l \Rightarrow t \in T_D(M, \neg R))$*

Sets $T_S(M, \neg R)$ and $T_D(M, \neg R)$, introduced section 3, respectively represent the set of successful attacks w.r.t. $\neg R$ and the set of detected attacks (with a call to function `atk_detected`). Our tool Lazart allows us to establish the protection level. First we compute $T_S(M, \neg R)$ starting from order 1 to a given bound max until an attack is found. On the contrary Lazart does not directly compute $T_D(M, \neg R)$. Then we will compute $T_S(M, R)$, the set of attacks verifying R and

not detected, that must be empty for 1 to $l-1$ faults (implying that all attacks are detected). Table 4 shows the protection level of our different countermeasures (listings 1.4, 1.5, 1.9) for the two fault models test inversion and data load modification and their combination (denoted Comb1 here).

Table 4. Protection levels of countermeasure schemes

Countermeasure \ Fault model	Test inversion	Load modification	Comb1
Test duplication	1	1	1
Load duplication	1	1	1
Block signature	1	1	1
i Test duplication	i	1	i
i Load duplication	i	i	i
i Block signature	i	i	i

Because all single fault models produce a protection level of 1 it is not mandatory to compute the protection level for the combination (combinations of models can sometimes allow to produce more efficient successful attacks, but it is not the case for 1). We extend our protection level when i new instances of countermeasures are considered, because duplicating countermeasures preserves the adequacy of protection scheme[2].

Discussion/limitation: Protection level is a partial function: it is not always possible to compute a protection level when the second condition does not hold. That means that this countermeasure against the considered fault models can not be analyzed in a modular way. For instance if we consider a fault model jumping anywhere in the code our countermeasures do not fulfill this condition. In this case countermeasures must embed local verification (for instance on the the value of the program counter).

5.2 Our methodology to harden programs

The approach we propose allows to harden a program P up to n faults, given as a set of fault models M and a success condition S. It relies on a *catalog* C of available countermeasures associated to the model for which they are *adequate* and their *protection* levels with respect to the fault models in M. This approach is based on the following steps:

1. Run Lazart to get the set of faults attacks $T_S(M, S)$ up to n. These attacks characterize the robustness of P with respect to M and S.
2. Compute the subset A of minimal attacks (definition 1) and the set IP of their associated injection points (section 2.3). IP gives a superset of the injection points to be protected in order to harden P.
3. Select both a subset $IP' \subseteq IP$ of injection points to protect, together with the corresponding countermeasures available in C.

[2] It is also possible to redefined new protection schemes combining or duplicating countermeasures and prove their adequacy as before.

4. Build a hardened program P' obtained by protecting each element of IP' using the results of step 2.

The question is now to decide whether P' is robust *by construction* up to n faults. Two sufficient conditions are given below (propositions 2 and 3) leading to the algorithm proposed in Section 5.3.

Proposition 2. *Let A a set of attacks obtained on a program P with respect to fault models M and a security objective S. Let P' obtained from P by protecting a **single** ip of each attack trace a of A with an adequate countermeasure of protection level $pl > n$. Then P' contains no longer any attack of A.*

To prove this proposition let us consider a k-faults attack $a = [ip_1, ip_2, \cdots, ip_k]$ of A. According to Section 5.1, faulting ip_i in P' requires now $pl+1$ faults, hence each attack of A does no longer succeed in less than n faults. Note that this proposition (trivially) holds when the countermeasures cannot be attacked (e.g, they are implemented using tamper-resistant hardware protections), since their protection levels can be considered as infinite.

Proposition 3. *Thanks to Definition 1 on minimal attacks, if we protect every minimal attacks up to n, all attacks in $T_S(M, S)$ will be protected up to n.*

5.3 Countermeasure placement algorithm

Algorithm 1 aims to protect one *ip* per execution trace using a countermeasure with a maximal available protection level. It iterates over the attacks of A by increasing order of faults injected. For each attack a, it builds first the set IP of injection points of a with an adequate countermeasure c with a minimal l greater than n. Then, it selects a candidate *ip* of IP with the most occurrences in the set of remaining attacks. If pl is equal to n then attack a becomes protected, together with a maximal number of other attacks sharing this injection point. Otherwise a is left unprotected. We denote by $(c, l) \in C(ip)$ a countermeasure and protection level of C of fault model related to the injection point *ip* (encoded into IP representation as exemplified in Section 2.3).

5.4 Formal results

We can now state results on Algorithm 1 based on Propositions 2 and 3.

Proposition 4. *Assurances supplied by algorithm 1*

1. *Algorithm1 terminates.*
2. *We have $P' \geq^{rob}_{n,I,m,S} P$ (attacks in Protected or with a prefix in Protected are no longer successful attack up to n and no new attack is introduced)*

Leveraging the result obtained from A to P needs A to be *representative* of P (i.e., each possible concrete attack of P is represented by a path in the set $T_S(M, S)$). In our case this means that the path enumeration performed by

Algorithm 1 Heuristic-based hardening algorithm

Require: a program P, a set of attack traces A of P up to a n faults, a catalog C
  $Protected \leftarrow \emptyset$ ▷ set of protected traces of A
  $IpProtected \leftarrow \emptyset$ ▷ set of protected injection points of A
  for k in 1 to n **do**
    $A_k \leftarrow$ attacks of A with k faults
    for a in $A_k \setminus Protected$ **do** ▷ for all not yet protected attacks
      $IP \leftarrow \{ip \in a \mid \exists (c,l) \in C(ip)$ s.t. c is adequate for ip and $l \geq n\}$
      if $IP \neq \emptyset$ **then** ▷ keep the ips of IP with a minimal protection level $\geq n$
        $IP \leftarrow \{ip \in IP \mid \exists (c,l) \in C(ip)$ s.t. $\forall ip' \in a.\ \forall (c',l') \in C(ip').\ l' > l\}$
      else ▷ take the ips of a with a maximal protection level
        $IP \leftarrow \{ip \in a \mid \exists (c,l) \in C(ip)$ s.t. $\forall ip' \in a.\ \forall (c',l') \in C(ip').\ l' < l\}$
      end if
      choose $ip \in IP$ with a maximal number of occurrences in $A \setminus Protected$
      if $pl + 1 > n$ **then**
        $IpProtected \leftarrow IpProtected \cup \{(ip, c)\}$ ▷ ip will be protected with c
        $Protected \leftarrow Protected \cup \{a' \in A \mid ip \in a'\}$
      end if
    end for
  end for
  $P' \leftarrow P$
  for all (ip, c) in $IpProtected$ **do**
    $P' \leftarrow$ protect ip of P' with c
  end for
  return $(P', Protected)$

Lazart to produce A is complete. Under this hypothesis, if $Protected = A$, then P' is robust up to n faults.

A degraded version. If, for some ip, C does not contain an adequate countermeasure with a protection level pl greater than $n - 1$ proposition 4 still holds when replacing n by x faults, where x is the minimal protection level used to protect traces of $A \setminus Protected$.

In the general case we cannot say anything beyond x faults, because injecting more than x faults on a countermeasure may lead to new execution states and hence new attacks in P'. In this case P' should be hardened as well by repeating the same procedure. This iterative process terminates either if we get rid of all attacks with strictly less than n faults, or if some k-faults attacks (for $k \leq n$) will always remain unprotected due to a lack of available countermeasure in C. Furthermore, for particular fault models (as test inversion and data load modification by any value) we could be able to show that fault injections on the protected scheme does not introduce new paths (showing that successful attacks for the level of protection does not produce states that are not covered by the associated mutation scheme). Then, if A is representative of P, deviant paths will be already analyzed[3].

[3] It is not the case for instance if our data load modification is reduced to increment the value which is read.

5.5 Experimentations

In this section, we give the results obtained with various countermeasure placement algorithms. We consider the test inversion and data load mutation fault models, with their combination, and the i Test duplication and i Load duplication countermeasures. Two programs are used for experimentations: *VP* the version 4 of the verifypin program in the FISSC benchmark [16] and *FU* an implementation of a firmware updater. *VP* is a generalization of our introductive examples (section 2) and all these programs are already be used for large experimentations [18] and are freely available[4].

Our results are summarized in Table 5. Columns PM gives the program to protect and the fault models to consider. Column IPs gives the total number of injection points of P with respect to m. We consider three placement algorithms:

- $Naive$: protection of level n for all injection points of P.
- All: protection of level n for all injection points involved in at least one successful attack
- $Single$: protection of a single injection point at level n per sucessful attack (Algorithm 1).

Column "#added-cms" indicates the number of countermeasures added, for each algorithm, up to $n = 4$ faults. As expected, Algorithm 1 gives much better results than the two other ones. In particular the $Naive$ one, used by all tools adding countemeasures in a systematic way (disregarding actual attacks) is clearly outperformed. Moreover, Algorithm 1 significatively reduces the number of added countermeasures, in particular on the FU example[5].

Table 5. Countermeasures added by placement algorithms

PM	IPs	algorithm	#added-cms 1 faults	2 faults	3 faults	4 faults
VP	8	Naive	8	16	24	32
with test inversion		All	3	8	12	16
		Single	3	6	9	12
FU	42	Naive	42	84	126	168
with Test inversion		All	0	28	42	88
		Single	0	14	21	28
FU	2	Naive	2	4	6	8
with data load mutation		All	1	4	6	8
		Single	1	2	3	4
FU	44	Naive	44	88	132	176
with test inversion		All	1	32	60	96
+ data load mutation		Single	1	16	24	32

[4] lazart.gricad-pages.univ-grenoble-alpes.fr/home/fdtc20/index.html
[5] still considering a relevant security property

6 Conclusion

In this paper we propose a global methodology assisting developer for hardening applications against multi-fault attacks, which is nowadays the state of the art [1,2]. Generalizing the existing single fault hardening approach is far from easy due to the he attack surface introduced by countermeasures, potentially introducing new attacks and new paths to explore. In order to master this inherent complexity we propose an innovative approach based on two properties: the *adequacy* of a countermeasure against a given fault model and the *protection level*, characterizing it weakness against a set of fault models (number of faults greater than the protection level could produce out of control behaviors). The proposed approach is limited to countermeasures with a local detection power, as for all automatic tools adding countermeasures in the single fault context.

At first we give a formal framework for specifying and proving fault models, countermeasures and protected schemes to establish the adequacy property. The key is the specification of the nominal behavior of instructions, how it can be impacted by faults and how deviations can be detected. Our work can be seen as a generalization of the approach described in [4], also based on the Frama-C/WP plugin, as explained in Section 4.3. Formally establishing the robustness of a countermeasure scheme against an attacker model is not new. In the context of Control flow integrity many countermeasures have been proposed and proved, as in [30]. In the context of fault injection, formal methods have been used to establish the effectiveness of countermeasures [22,23]. But these works are dedicated to particular form of countermeasures and specific fault models, and they address single faults only.

Based on the new notion of *protection level* we propose algorithms allowing to automatically insert countermeasures, going beyond automatic algorithms not taking into consideration successful attacks (the naive approach). Furthermore we are able to characterize the level of assurance of the protected programs (property 4). Nevertheless, computing which injection points must be protected, w.r.t. successful properties requires a first analysis giving some exhaustive insurances. Here we based this analysis on an symbolic execution engine (Klee [17] and Lazart [15]), which is sensitive to the code complexity (number of paths and number of fault injections). We are currently exploring several ways for mastering this complexity based on static analysis [19] and a smart encoding of faults [31], which could be combined with the approach proposed here.

As pointed out, a such generic framework is new and must be more largely evaluated, considering existing fault models and countermeasures, together with more sophisticated countermeasures and more flexible placement algorithms. In particular we are studying an algorithm, based on Integer Linear Programming, allowing to combine several countermeasures per attacks, with lower protection levels, still preserving our robustness properties.

Acknowledgment

This work is supported by SECURIOT-2-AAP FUI 23 and by the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-15-IDEX-02) and the project ARSENE of PEPR Cybersecurité.

References

1. C. H. Kim and J. Quisquater, "Fault attacks for CRT based RSA: new attacks, new results, and new countermeasures," in *Information Security Theory and Practices. WISTP 2007, Heraklion, Greece, 2007, Proceedings*, 2007, pp. 215–228.
2. E. Trichina and R. Korkikyan, "Multi fault laser attacks on protected CRT-RSA," in *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2010, Santa Barbara, California, USA, 21 August 2010*, 2010, pp. 75–86.
3. ANSSI, Amossys, EDSI, LETI, Lexfo, Oppida, Quarkslab, SERMA, Synacktiv, Thales, and T. Labs, "Inter-cesti: Methodological and technical feedbacks on hardware devices evaluations," in *SSTIC 2020, Symposium sur la sécurité des technologies de l'information et des communications*, 2020.
4. T. Martin, N. Kosmatov, and V. Prevosto, "Verifying redundant-check based countermeasures: a case study," in *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing*. ACM, 2022.
5. M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer bug to gain kernel privileges: how to cause and exploit single bit errors," in *Black Hat*, 2015.
6. D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *DIMVA 2016, San Sebastiá*, 2016, pp. 300–321.
7. A. Tang, S. Sethumadhavan, and S. J. Stolfo, "CLKSCREW: exposing the perils of security-oblivious energy management," in *26th USENIX Security Symposium, USENIX Security 2017*. USENIX Association, 2017.
8. P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM, 2019.
9. K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against Intel SGX," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.
10. N. Timmers and A. Spruyt, "Bypassing secure boot using fault injection," in *Black hat Europe 2016*.
11. S. Delarea and Y. Oren, "Practical, low-cost fault injection attacks on personal smart devices," *Applied Sciences*, vol. 12, no. 1, 2022.
12. C. Hillebold, "Compiler-assisted integrity against fault injection attacks," Master's thesis, University of Technology, Graz, December 2014.
13. J. Proy, K. Heydemann, A. Berzati, and A. Cohen, "Compiler-assisted loop hardening against fault attacks," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, pp. 36:1–36:25, 2017.
14. N. Belleville, K. Heydemann, D. Couroussé, T. Barry, B. Robisson, A. Seriai, and H. Charles, "Automatic application of software countermeasures against physical attacks," in *Cyber-Physical Systems Security.*, 2018, pp. 135–155.

15. M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections," in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014*. IEEE, 2014, pp. 213–222.
16. L. Dureuil, G. Petiot, M. Potet, T. Le, A. Crohen, and P. de Choudens, "FISSC: A Fault Injection and Simulation Secure Collection," in *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016*, 2016, pp. 3–11.
17. C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, USA*, 2008, pp. 209–224.
18. E. Boespflug, C. Ene, L. Mounier, and M.-L. Potet, "Countermeasures Optimization in Multiple Fault-Injection Context," in *"Fault Diagnosis and Tolerance in Cryptography" FDTC 2020*, Sep. 2020.
19. G. Lacombe, D. Feliot, E. Boespflug, and M.-L. Potet, "Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities," *Journal of Cryptographic Engineering*, vol. 18 january 2023.
20. M. Christofi, B. Chetali, L. Goubin, and D. Vigilant, "Formal verification of a CRT-RSA implementation against fault attacks," *Journal of Cryptographic Engineering*, vol. 3, no. 3, pp. 157–167, 2013.
21. P. Rauzy and S. Guilley, "A Formal Proof of Countermeasures Against Fault Injection Attacks on CRT-RSA," *Journal of Cryptographic Engineering*, vol. 4, no. 3, pp. 173–185, 2014.
22. L. Goubet, K. Heydemann, E. Encrenaz, and R. De Keulenaer, "Efficient Design and Evaluation of Countermeasures against Fault Attack with Formal Verification," in *14th Smart Card Research and Advanced Application Conference, CARDIS*, Nov. 2015.
23. K. Heydemann, J.-F. Lalande, and P. Berthomé, "Formally verified software countermeasures for control-flow integrity of smart card c code," *Computers & Security*, vol. 85, pp. 202–224, 2019.
24. T. CCRA Management Committee. (2012, Sep.) Common Criteria for Information Technology Security Evaluation. [Online]. Available: http://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R4.pdf
25. "Application of Attack Potential to Smartcards and Similar Devices," Joint Interpretation Library, Tech. Rep. Version 3.0, June 2020.
26. L. Dureuil, M.-L. Potet, P. d. Choudens, C. Dumas, and J. Clédière, "From code review to fault injection attacks: Filling the gap using fault model inference," in *14th Smart Card Research and Advanced Application Conference, CARDIS15*. LNCS, 2015.
27. N. Theissing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 404–409.
28. N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, and E. Encrenaz, "Experimental evaluation of two software countermeasures against fault attacks," in *IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2014*.
29. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Asp. Comput.*, vol. 27, no. 3, pp. 573–609, 2015.
30. M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

31. S. Ducousso, S. Bardin, and M.-L. Potet, "Adversarial reachability for program-level security analysis," in *European Symposium on Programming (ESOP)*, 2023.

A Frama-C schemes

A.1 Mutation schemes

Data load mutation. We can mutate a load with the right value, introducing in that a fault with no dangerous impact.

```
/*@ ensures  assigns \nothing ;@*/
int choose (int value);
/*@ assigns \nothing ;
  @ behavior nominal :
  @    assumes  fault==0 ;
  @    ensures  \result==value;
  @  behavior faulted :
  @    assumes  fault!=0 ;
  @    ensures  \true ; @*/
int mutation_scheme_dl(int value, int fault)
   { int x = value; if(fault) x = choose(value); return x; }
```

Listing 1.7. Data load mutation scheme

Else following Then mutation. We encode the flow by 0x10 when we cross the block Then, by 0x01 when we cross the block Else and by 0x11 when we cross the block Then followed by the block Else.

```
  /*@ behavior nominal :
       @ assumes fault==0  ;
       @ ensures  ((C!=0 ==> \result == 0x10)) ;
       @ ensures  ((C==0 ==>  \result == 0x01)) ;
    @ behavior faulted :
       @ assumes fault!=0  ;
       @ ensures  ((C!=0 ==> \result == 0x11)) ;
       @ ensures  ((C==0 ==> \result == 0x01)) ;@*/
  int mutation_scheme_then_else(int C,  int fault)
  { int FLOW = 0x00;
    if(C) goto then_br;  else goto else_br;
    then_br : FLOW|=0x10; if(!fault) goto fin;
    else_br : FLOW|=0x01;
    fin : return FLOW; }
```

Listing 1.8. Else following Then mutation scheme

A.2 Countermeasure schemes

```
/*@  ensures \result==value;
  @ assigns \nothing ; @*/
void LoadDup(int res, int value)
   {int cm_var  = value; if(res != cm_var) atk_detected();}
```

Listing 1.9. Load duplication countermeasure scheme

A.3 Protected schemes

```
/*@ behavior nominal :
  @ ensures \result==value; @*/
int protected_DL_LD(int value, int fault){
    int res = mutation_scheme_dl(value,fault);
    LoadDup(res,value); return(res); }
```

Listing 1.10. Load modification protected by load duplication

```
/*@ assigns RTS;
  @ behavior nominal :
  @    ensures  (C!=0 ==>  \result==br_then) ;
  @    ensures  (C==0 ==>  \result==br_else) ; @*/
int protected_EfT_SIG(int C, int br_then, int br_else, int fault){
    sign(C, br_then, br_else);
    if((C && !fault) || (!C && fault)
        { check(br_then); return br_then; }
    else { check(br_else); return br_else; } }
```

Listing 1.11. TI protected by Signature

```
  /*@  requires id_true!=id_false ;
  @ assigns RTS ;
  @ behavior nominal :
  @     ensures  ((C!=0  ==> \result == 0x10)) ;
  @     ensures  ((C==0  ==>  \result == 0x01)) ;@*/
  int protected_ThenElse_CMSign(int C, int id_true, int id_false, int fault)
  { int FLOW = 0x00;
    sign(C,id_true,id_false);
    if(C) goto then_br; else  goto else_br;
    then_br : check(id_true); FLOW|=0X10; if(!fault) goto fin;
    else_br : {check(id_false); FLOW|=0X01;}
    fin : return FLOW; }
```

Listing 1.12. Else following Then protected by Signature

```
  /*@ behavior nominal :
  @     ensures  ((C!=0 && fault==0 ==> \result == THEN)) ;
  @     ensures  ((C==0  ==>  \result == ELSE)) ;@*/
  int protected_ThenElse_CM_Testdup(int C, int fault)
  { int FLOW = 0x00;
    if(C) { CM_TestDup(C); goto then_br; } else { CM_TestDup(!C); goto else_br;
          }
    then_br : FLOW|=0x10; if(!fault) goto fin;
    else_br : FLOW|=0x01;
    fin : return FLOW; }
```

Listing 1.13. Else following Then protected by Test duplication