A compositional methodology to harden programs against multi-fault attacks

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

A compositional methodology to harden programs against multi-fault attacks

Etienne Boespflug, Laurent Mounier, Marie-Laure Potet, Abderrahmane Bouguern
VERIMAG / Université Grenoble Alpes
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 by adding countermeasures. According to the state of the art, applications must now be protected against *multi-fault injection* [23], [33]. As a consequence developing applications which are robust becomes a very challenging task, in particular because countermeasures can be also the target of attacks [4], [20]. 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 and place the appropriate countermeasures, giving guarantees on the robustness of the protected program.**

***Index Terms*—multiple fault-injection; code analysis; software countermeasure; dynamic-symbolic execution**

I. 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 targeted 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 [30], or by exploiting some weaknesses in the power management modules [27], [22]. 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 [32], [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* [23], [33], 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 trial and errors game, in particular because countermeasures can be also the target of attacks [4], [20]. The aim of this paper is to propose a methodology for developers allowing to harden an application against multi-fault attacks: taking into account the fact that countermeasures themselves can be attacked and helping to place protection schemes inside the program w.r.t a set of fault models and an attack objective.

There exist tools adding countermeasures, generally at compilation-time. They are dedicated to particular fault models (data modification, instruction skip, flow integrity) [17], [26], [5], for instance by adding redundant checks or duplicating idempotent instructions. Nevertheless these tools generally 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-fault must be taken into account, since more countermeasures must be added, potentially adding unnecessary performance/size overheads.

The objective of this paper is to assist security developers in the software countermeasure insertion process in proposing a compositional approach. More precisely this paper provides the following contributions:

1) we formulate the problem of robustness comparison of programs in presence of multi-fault;
2) we propose a methodology to analyze countermeasures "in isolation", studying their effectiveness at blocking attacks in single-fault and multi-fault contexts;
3) we propose several algorithms allowing to harden applications starting from identified attacks, based on the previous isolation analysis and we establish their guarantees w.r.t. the proposed comparison definition;
4) we provide an implementation of this approach, based on a Dynamic Symbolic Execution (DSE) tool and evaluate our approach on a benchmark of code examples.

Section II introduces multiple fault-injection and countermeasures through a motivating example and presents the tool Lazart. Section III proposes definitions for robustness evaluation and comparison dedicated to multi-fault. Section IV describes the proposed methodology for analyzing countermeasures in terms of their adequacy and weakness against established fault models. Section V presents our countermeasure placement algorithms and illustrates them on several examples. Finally, Section VII discusses related works, limitations of our solution and future directions.

II. Multi-Fault attacks and defences

Firstly we briefly present Lazart, a tool allowing to analyze high level code against multi-fault attacks, which has been successfully used for assisting developers and auditors [4], [18], thanks to its combination of multi-fault capabilities and expressive fault models.

A. *łazart: a symbolic multi-fault injection tool*

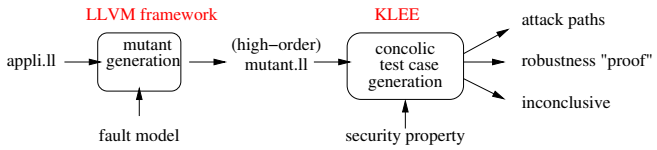

Fig. 1. Lazart workflow

Lazart [25], [6], is a multi-fault analysis tool based on the KLEE DSE engine [7] which computes multi-fault injection attack paths (see Fig. 1). This is achieved by mutating the target program (at the LLVM level) in order to simulate faulty behaviors based on some fault models and performing a dynamic symbolic execution analysis to find execution paths violating a security property. If no such paths are obtained, and if the path exploration is complete, the target program is robust. Otherwise KLEE exhibits all different paths violating the security property, supplying in that a coverage of attacks. Inconclusive verdicts are issued when the full path coverage cannot be reached. Lazart supports the two main fault models, that can be used to simulate other more complex models very effectively:

- **Test Inversion** where a conditional jump can be inverted (e.g., in *if* instructions and loop conditions).
- **Data Fault** where the value returned when reading a variable from memory can be altered (the mutated value being made symbolic, possibly with extra constraints).

```
// a sensitive conditional jump
if (C) then goto bb_true else goto bb_false;

// the mutation due to a fault
klee_make_symbolic(inject);
if (inject && fault_nb<n)
{
    fault_nb++;
    if (C) goto bb_false; else goto bb_true;
}
else if (C) goto bb_true; else goto bb_false;
```

Listing 1. Test inversion mutation

Listings 1 and 2 illustrate how sensitive instructions are mutated, transformations being described at the C level for readability reasons. The function `make_symbolic` produces a symbolic variable for the symbolic execution engine KLEE: two values for the variable `inject` will be produced in order to execute the two branches of the mutation schemes (without fault, and with fault, if the fault limit `n` is not exceeded). In the same way a symbolic value is chosen in Listing 2, possibly constrained by a predicate `Px` supplied by the user. Thanks to the symbolic execution engine and because a single mutant is produced embedding all possible faults, the combinatorial path exploration process is mastered. In particular, symbolic data mutation could result in the exploration of several paths, giving in that a very powerful fault model.

```
// a sensitive data load operation
v = x;

// the  mutation due to a fault
klee_make_symbolic(inject);
if (inject && faults_nb<n)
{
    klee_make_symbolic(new_x);
    klee_assume(Px(new_x);
    faults_nb++;
}
else {new_x = x;}
v= new_x;
```

Listing 2. Symbolic data load mutation

B. *Two versions of a byteArrayCmp function*

We now introduce an example illustrating how a fragile version can be hardened. Listing 3 is an excerpt of a `byteArrayCmp` function used in the verifyPIN collection taken from the FISSC public benchmark [12]. 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 and only if the two PIN codes are equals.

```
1 BOOL byteArrayCmp(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             result = BOOL_FALSE;
7     return result;
8 }
```

Listing 3. Fragile byteArrayCmp function

This example is insecure against fault injection consisting in inverting control flow conditions. Table I line V1 gives the results supplied by Lazart for Listing 3 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.

TABLE I
Lazart results for the `byteArrayCmp` versions

# faults	0	1	2	3	4	5	6	7	8
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), 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.

Listing 4 presents a more robust version of the function `byteArrayCmp`, adding classical redundant-check countermeasures: 1) a test line 9 checking the exit value of the loop

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

counter and 2) a systematic duplication of the verification of the result of conditions (lines 6 and 8). Calls to the function `atk_detected` stop the execution, signaling the detection of an abnormal behavior. The attacker is now able to bypass countermeasures. For instance here he can invert the extra tests (lines 6, 8 and 9) in order to hijack the detection. Results supplied by Lazart for this new version (still for the test inversion fault model) are given on Table I, 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).

```
1  BOOL robust_byteArrayCmp(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          }
9          else if(a1[i] != a2[i]) atk_detected();
10     }
11     if(i != size)  atk_detected();
12     return result;
13 }
```

Listing 4. Example: A robust byteArrayCmp function

TABLE II
LAZART HOTSPOTS ANALYSIS FOR LISTING 4

IP line \ # faults	0	1	2	3	4	5	6	7	8	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**

Attacks are represented by sequences of fault occurrences consisting in pairs (injection point, fault model), where an injection point denotes an instruction occurrence which is sensitive to the fault model. For instance the first attack of Table I is represented by <IP4(TI), IP9(TI)>, where IP4 is the fault injection point of line 4 and TI denotes "test inversion". From that Lazart can compute how many times a fault injection point occurs into successful attacks. Table II shows results associated to Table I, for the robust version of *byteArrayCmp* (listing 4). In particular, we can observe that attacking IP6, corresponding to an added countermeasure, never results in a successful attack. As a consequence this countermeasure is superfluous for the considered security property. In the same way if we only consider attacks of order 2 or 3, countermeasures lines 5 and 8 are no longer useful.

C. Our objectives and contributions

In single-fault context, hardening a code generally follows a "try and test" approach: countermeasures are added as long as attacks are found and robustness is checked again through a new analysis (as such a secure implementation is now expected to become robust against single-fault attacks). However, this approach becomes impractical in a multi-fault context since countermeasures can be themselves targeted by attacks. The aim of this paper is to propose several placement algorithms allowing to add countermeasures advisedly, targeting to avoid unnecessary code, as illustrated before. 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, countermeasures allowing to detect errors in data or control flows, at software level.

To the best of our knowledge, this approach is innovative. As pointed out before, tools generally add countermeasures in a systematic way (as *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 solution introduces serious and unnecessary run-time overheads. On the other hand, methods have been proposed to prove hardened programs [8], [28], [20], or to verify the efficiency of a given form of countermeasures against attacker models [15], [16]. All these approaches are developed in the context of single fault and adapted to a particular countermeasure or fault model. On the contrary the approach we consider here is general, modular, able to cope with multi-fault attacks and addressing the problem of optimization of countermeasures placements The methodology presented in [6] proposes to reduce the set of countermeasures applied in a program already protected, without introducing new faults. The approach we propose here allows to add countermeasures on a program (that may be already partially protected) in order to ensure a robustness in n faults for a set of fault models.

III. ROBUSTNESS METRICS FOR MULTI-FAULT

In the context of certification for high levels of security [9], applications and devices are submitted to vulnerability analyses (the AVA class of Common Criteria), conducted by expert teams (as ITSEF laboratories). For instance rating physical attacks for smart cards or similar devices is established by the JIL Hardware Attacks Subgroup [2], based on a set of relevant factors: elapsed time, expertise, knowledge of the target of evaluation, access to it, ... etc.

Concerning code-based simulation tools some metrics are generally used such as the number of successful attacks, potentially weighted by the total number of attack or the attack surface [13]. Nevertheless these metrics are not really used in practice to compare robustness of applications: the "try and test" approach turns out to be sufficient when single fault is considered. On the contrary, in the context of multi-fault we have to formalize how implementations can be compared, as we propose here. Furthermore we will use our formalization

to validate our methodology and algorithms showing that we improve the robustness of applications when adding countermeasures.

A. 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 nominal execution steps or faulted steps, starting from an initial state. In the following $init(t)$ denotes the initial state (including inputs) 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 ($fault(t) = 0$)
- $T_D(M,S)$: faulted traces that are detected by a countermeasure
- $T_S(M,S)$: successful attacks (faulted traces verifying S and not detected)
- $T_F(M,S)$: non detected faulted traces that do not verify S

Traces terminating in error can be assimilated to one of these classes by the user, depending if he considers them as a form of attack detection, a successful attack or a failure.

Definition 1: Input vulnerability characterization

Let P be a program, i an initial state, μ the order to consider, M a set of fault models and S a success 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:

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

Definition 1 can be extended to a set of initial states I. For instance Table I 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 being `result==BOOL_TRUE`.

Definition 2: Robustness level

If I is the set of all initial states and $Vuln_\mu(P,i,M,S)(n) = 0$ for all $n \in [0..\mu]$ and for all $i \in I$ we say that P *is said to be robust up to* μ *faults*.

B. Robustness comparison

Here we target to compare a program P with a hardened version of P. For instance we want to state that the secure version of function `byteArrayCmp` is more robust than the initial version, up to order 8. We propose a fine grained definition, respecting transitivity and, more importantly, a monotony property.

Definition 3: Input robustness comparison

Let P be a program, i an initial state, 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:

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

As before, this definition can be extended to a set of inputs, and we define $P' \geq^{rob}_{\mu,M,S} P$ if the condition holds for each possible inputs.

Our definition does not only compare the total number of attacks for a given order but all intermediary sums, telling we expect a hardened version to increase the number of faults required to get successful attacks. For instance for the example of section II attacks of order n are moved to order $2 * n$ (Table I). Thanks to definition 3, several nice properties hold as *monotony* ($P' \geq^{rob}_{n+1,i,M,S} P \Rightarrow P' \geq^{rob}_{n,i,M,S} P$) and *transitivity* ($P'' \geq^{rob}_{\mu,i,M,S} P' \wedge P' \geq^{rob}_{\mu,i,M,S} P \Rightarrow P'' \geq^{rob}_{\mu,i,M,S} P$).

C. Discussion

A classical approach to evaluate the robustness of a code against fault attacks 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 [31], [21], [13] 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-fault 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, taking in that the attack surface paradox [13].

IV. Countermeasures analysis

We will consider *detection countermeasures* (based on a potential proper internal state and some verifications allowing to detect an attack) for which the protection scheme is applied on an Injection Point (IP) for a fault model m. This section proposes a systematic approach for analyzing such countermeasures "in isolation", aiming to compare the behavior of *Unprotected Schemes* and *Protected Schemes* in multi-fault contexts:

1) **Sensitive Schemes** (*SS*), which characterize structures and instructions sensitive to a fault model (i.e. an IP for the model m), and its nominal behavior.
2) **Protected Schemes** (*PS*), which describe how countermeasures are inserted to protect Sensitive Schemes (SS) to produce a protected IP.

The targeted goal is to replace sensitive schemes with adapted protected schemes depending on their properties. We will illustrate our approach using two fault models: *test inversion* and *symbolic data load modification*. Other forms of

fault models and countermeasures are discussed at the end of this section.

A. Sensitive schemes and protected schemes

We illustrate PS and SS with two countermeasures: *test duplication* and *load duplication*. Listing 5 encodes the LLVM instruction `br %cond bb_true bb_false` corresponding to a SS for model *test inversion* and the PS *test duplication*. In the same way, Listing 6 describes the LLVM statement `%target = load %var` (SS for symbolic data load IP) and the PS *load duplication*. A call to the function `atk_detected` corresponds to the detection of an attack. Listings 5 and 6 are stated at the C level, for readability issue, but Fig. 2 and 3 give the actual LLVM implementation of these protections and the corresponding SS. The red and purple circles corresponds respectively to test inversion IPs and to data load mutation IPs. The yellow code blocks corresponds to SS's basic blocks that are modified by the protection scheme. The gray blocks corresponds to unmodified basic blocks and blue blocks to new basic blocks added by the PS.

```
1  // sensitive scheme
2  int cond_scheme(int cond, int bb_true, int bb_false) {
3      if(cond) return bb_true;
4      return bb_false;
5  }
6
7  // protected version
8  int test_duplication(int cond, int bb_true, int bb_false
       ) {
9      if(cond) {
10        if(!cond) atk_detected();
11        else return bb_true;
12     }
13     if(cond) atk_detected();
14     return bb_false;
15 }
```

Listing 5. test inversion sensitive scheme and its protected version

```
1  // sensitive scheme
2  int data_load(int *var) {
3      int target = *var;
4      return target;
5  }
6
7  // protection version
8  int load_duplication(int *var) {
9      int target = *var;
10     int clone  = *var;
11     if(target != clone) atk_detected();
12     return(target);
13 }
```

Listing 6. load sensitive scheme and its protected version

We are now introducing two properties that will be used later in our placement algorithms : *the Adequacy condition* stating that a protected scheme detects faults of a given fault model, in single-fault scenario, and the *Vulnerability level*, computing the "resilience" of a protected scheme against a set of fault models in multi-fault context.

B. Adequacy condition

Definition 4: Adequacy condition

Let m be a fault model associated to a sensitive scheme SS and PS a protection scheme associated to SS, i.e. targeting to detect an attack with the fault model m. SS can be safely replaced by PS in a program P iff:

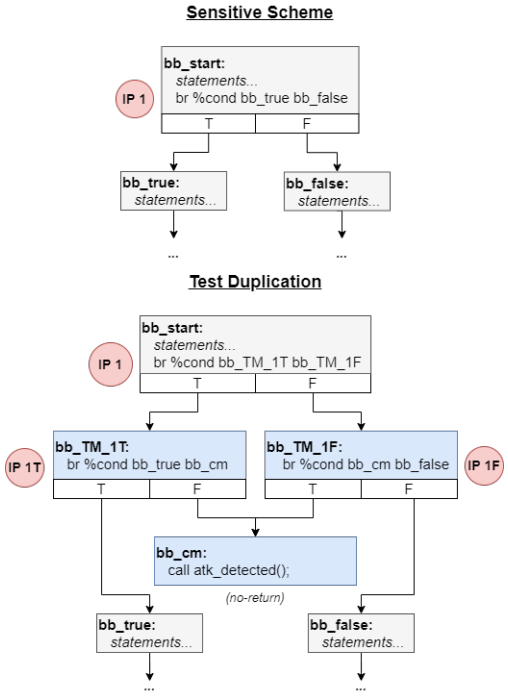

Fig. 2. Test duplication in LLVM and its attack surface.

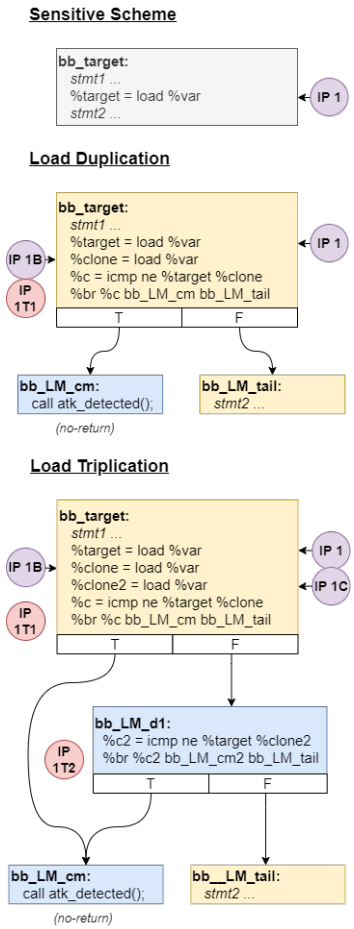

Fig. 3. Load sensitive scheme and two protected versions

- **condition 1:** in absence of a fault of m PS behaves as SS (or possibly stops before the end of the scheme)
- **condition 2:** in the presence of a fault of m either PS behaves as SS (the fault has no impact), or the attack is detected (through a specific function `atk_detected`)
- **condition 3:** PS does not modify any variables of P in other way than SS (frame condition)

If the 3 conditions above are fulfilled we can establish that PS is robust for m up to 1 fault.

For instance we will prove that the protected version of Listing 3 is secured against the effect of a test inversion fault described on Listing 1, and similarly data modification (Listing 3 provides a protected scheme against mutation described in Listing 2).

Definition 4 describes a set of sufficient conditions which can be reasonably verified in practice. For condition 3 a manual frame analysis can be conducted, but generally protection schemes do not change any variable except their internal ones. For conditions 1 and 2 we run Lazart twice on the protection scheme, applying the fault model m only on the injection point already present in the sensitive scheme, with two configurations. Firstly we use no oracle in order to cover all feasible paths of PS for any input value. Let C be the set of obtained traces. Secondly we run Lazart with an oracle specifying the expected nominal behavior of SS. Let N be the set of obtained traces. Then we verify that all paths in $C - N$ terminate in a countermeasure state (calling the `atk_detected` function). That means that it does not exist any 1-fault attack violating the nominal behavior (this case corresponding to an erroneous countermeasure) nor reaching any nominal exit point (e.g., because of some control flow hijacking). Such situations would correspond to a countermeasure which cannot be analysed in isolation[2]. Due to the simplicity of the analyzed code we can suppose here (and manually verify) that all paths have been covered.

Listing 7 illustrates how these verifications are conducted for our examples, lines 5 and 11 respectively stating the expected nominal behaviors. Parameters are made symbolic, in order to cover all paths[3]. The `klee_assume(P)` allows to only focus on paths verifying the predicate `P`.

```
1  int verify_ti_protection() {
2    int cond, bb_true, bb_false;
3    klee_make_symbolic(&cond, sizeof(cond), "cond");
4    int br = test_duplication(cond, bb_true, bb_false);
5    klee_assume((br == (cond ? bb_true : bb_false)));}
6
7  int verify_ld_protection() {
8    int value;
9    klee_make_symbolic(&value,sizeof(value),"value");
10   int result = load_duplication(value);
11   klee_assume(result == value); }
```

Listing 7. Adequacy protection verifications

[2]for instance if we hide the line `if (false) {int *p=null; return(*p)}` in a protection scheme it does not fit our condition 2. We can also hide the call to a malicious function which does not return.

[3]for readability reason we do not make symbolic the branches `bb_true` and `bb_false` in the function `verify_ti_protection`, and we suppose these two addresses are different.

TABLE III
ADEQUACY CONDITION FOR TEST INVERSION AND LOAD DUPLICATION

	#feasible paths	#paths in cm	#nominal paths	difference
TD	4	2	2	0
LD	3	1	2	0

Results relatively to conditions 2 and 3 are shown in Table III. Normally all paths with 0 faults are in N. After fault injections the protected scheme `test_duplication` (`TD` in table III) produces 8 paths among them 4 are unfeasible (terminating in cm with 0 faults or terminating in the nominal state with 1 fault). 2 remaining paths terminate normally (with 0 faults) and 2 are detected (with 1 fault)[4]. In the same way after a fault injection consisting in modifying the value which is actually loaded by any other value, the protected scheme `load_duplication` (`LD` in table III) produces 4 paths, among them 3 are feasible: a detected path when the new value differs from the initial value and two nominal paths when no fault is introduced or when the chosen value is identical to the initial value (a fault without effect).

C. Countermeasure vulnerability level

In the context of multi-fault, countermeasures come with their proper attack surface, the added code that may contains additional IPs. Thus, injecting faults into this code can produce new execution traces, which may potentially lead to successful attacks. To address this we define (and compute) the *vulnerability level* of a PS, which quantifies the minimal number of faults allowing to produce an abnormal behavior (w.r.t the nominal behavior of the corresponding SS) without detection, w.r.t a set of fault models.

Definition 5: Vulnerability level
Let PS be a protected scheme defined by its nominal behavior N and M a set of fault models including the model m for which PS is adequate. Let $vl(PS, M) = v$ the level of vulnerability of PS w.r.t. M defined as:

- **condition 1:** v is the minimal number of necessary faults producing an attack violating N: $v = min\{(fault(t) \mid t \in T_S(M, \neg N)\}$ (or ∞ if no successful attack exists).
- **condition 2:** any traces t such that $0 \leq fault(t) < v$ are detected or terminate in a nominal state: $\forall t \ (0 \leq fault(t) < v \Rightarrow t \in T_D(M, \neg N) \cup T_S(M, N))$

Sets $T_S(M, \neg N)$ and $T_D(M, \neg N)$, introduced in Section III, respectively represent the set of successful attacks w.r.t. $\neg N$ and the set of detected attacks. Condition 2 of Definition 5 is similar to Condition 2 of Definition 4: then we can verify this condition in iterating the previous process from 0 faults and, as soon as this condition is violated, we have reached the vulnerability level. Remark that a scheme with a vulnerability level of 1 cannot be adequate for the considered fault models,

[4]if the two adresses `bb_true` and `bb_false` are symbolic two new nominal paths appear because if these values are equal the fault has no impact on the nominal behavior.

TABLE IV
VULNERABILITY LEVELS OF COUNTERMEASURE SCHEMES

Countermeasure \ Fault model	Test inversion	Load modification	Comb
Test duplication	2	-	2
Load duplication	-	2	2
Load triplication	-	3	3
Test i-plication	$i+1$	-	$i+1$
Load i-plication	-	$i+1$	$i+1$

and an unprotected IP (SS) has a vulnerability level of 1 by definition.

Table IV shows the vulnerability level[5] of countermeasures of Figures 2 and 3, for the two fault models test inversion and data load modification and their combination (denoted Comb here). Figures 2 and 3 show the new injection points introduced in the test duplication scheme and data duplication and triplication. Here we extend the protected schemes Test duplication and Load duplication as new protected schemes corresponding to the i-plication of the protection[6], for which we can also compute the vulnerability level.

For a given set of fault models M, the "isolation" analysis allows for the generation of a catalog associating each PS with the fault model (in M) for which it is adequate, and its vulnerability level for M. This catalog C is used by placement algorithms presented in section V.

D. Discussion

We have presented here two basic detection countermeasures in particular because our implementation is actually tailored for them. But the methodology proposed here can be extended to some more complex sensitive and protected schemes: for instance an if-then-else structure such that the "else" part can be executed after the "then" part (a classical fault model consisting in nop-ing the goto instruction at the end of the "then" block) or classical CFI protections, for instance based on basic block signatures as SWIFT [29], [10] or instruction counter [16]. For this type of protection a global state is introduced such as a General Signature Register or a global instruction counter. Then the difficulty is first to specify properly the nominal behavior including this global state but also the precondition on the initial value of this global state. Secondly these classical countermeasures, dedicated to a 1-fault analysis, must be reinforced to be resistant to several fault and fault models (for instance the global state must be duplicated to be resistant to a load attack). Our methodology can become more complex for sophisticated protection such as loop protection [26], but this complexity is inherent to the proof process.

Some, fault models such as jumps are difficult to analyze in isolation, because they can introduce jump from and into the IP. Verifying pre/post-condition on the SS and PS using such fault models is part of future work.

[5]The symbol "-" denotes that the countermeasure cannot be attacked with this fault model. For test duplication, no attack surface exists for load mutation model. In the case of load duplication, tests can be inverted but cannot be successful without a combination with data load mutation.

[6]Load triplication protection scheme is presented in figure 3, and corresponds to the Load i-plication of level 2.

V. COUNTERMEASURES PLACEMENT ALGORITHMS

A. Objective and general approach

Our objective is to harden a program P, up to n faults, with respect to a set of fault models M and a given attack objective S. We want to generate a new program P' derived from P by protecting some of its sensitive IPs, thanks to a *catalog* C of available protection schemes. As defined in the previous section a catalog associates a protection scheme with its vulnerability level vl (w.r.t. M and n) and the fault model for which it is adequate.

In the following we propose several countermeasure placement algorithms and we show that, under some hypotheses, they allow to produce a program P' which is indeed robust up to n faults, according to Definition 2. Table V presents the principle of these five placement algorithms. All algorithms return a mapping, denoted as "Minimal Vulnerability Levels" (MVL), associating to each injection point ip of P, the minimum required vulnerability level, according to this algorithm[7]. Algorithm 1 describes how the hardened program P' is generated from mapping MVL. We suppose here the catalog C *complete with respect to* MVL, namely, for each ip, if $MVL[ip] = p$ then it exists a protection scheme ps in C such that ps is adequate for ip and with a vulnerability level of k with $k \geq p$ for the considered fault model set M. This hypothesis is valid for us because we dispose of "unbounded" protection schemes (Test i-plication and load i-plication) against test inversion and data load modification fault models. We discuss in section V-F what happens when this hypothesis is relaxed.

Algorithm 1 Generation of P'

Require: a program P, a set of minimal vulnerability level MVL, a catalog C
$P' \leftarrow P$
for all ip of P **do**
  if $MVL[ip] > 1$ **then** ▷ ip should be protected
    choose ps in C adequate for ip with a vulnerability level $> MVL[ip]$
    $P' \leftarrow$ protect ip of P' with ps
  end if
end for
return P'

B. Expected properties

Due to the definition of countermeasure adequacy (definition 4), P' either behaves as P or stops. We now want to establish that P' is robust at order n (according to definition 2 of section III) or, at least, more robust than P (according to definition 3 of section III).

[7]Note that $MVL[ip] = 1$ when ip can be left unprotected.

TABLE V
PRINCIPLE OF EACH PLACEMENT ALGORITHMS

Algorithm	Description
`naive`	All IPs in P are protected with $vl \geq n+1$.
`atk`	All IPs in attacks are protected with $vl \geq n+1$.
`min`	All IPs in minimal attacks are protected with $vl \geq n+1$.
`block`	At least one IP per minimal attacks is protected with $vl \geq n+1$.
`opt`	Protection is distributed between the IPs in minimal attacks, to get rid of attacks in less than $n+1$ faults.

Proposition 1 (Robustness of P'): The two following condition are sufficient to ensure the robustness of P':

- **condition 1:** all n-successful attack paths of P are detected in P';
- **condition 2:** P' should not introduce **new** attack path at order less or equal than n.

For instance the *naive* placement algorithm (Table V) protects all IPs in the program P with a vulnerability level of at least $n+1$. This algorithm generalizes, in the context of robustness of order n, the approach implemented in many systematic countermeasures placement tools [24], [5], [19] against 1-fault. Proposition 1 holds in this case since no injection point of P can be bypassed in less than $n+1$ faults (whatever is the attack objective S). The validity of conditions 1 and 2 will be discussed for each other countermeasure placement algorithm we propose below.

C. Systematic placement algorithms

The four last algorithms of Table V work from a computed set A of successful attacks on program P w.r.t. to M and S. We are interested to cover all attacks a of $T_s(m,S)$ with $fault(a) \leq n$, as computed in our examples of section II. We qualify the attack set A as *representative* when each trace of $T_s(m,S)$ owns a representative in A, meaning that each combination of faults leading to a successful attack $T_s(m,S)$ is represented in A. In particular symbolic execution is a good way to compute that[8]. In the following we suppose that A is representative and we discuss later (see section V-F) which guarantees can be stated when this hypothesis is relaxed.

1) Attack-based placement algorithm: This algorithm, called `atk` in Table V, protects all injection points of A at level $n+1$, namely;

$\forall ip \in IP(P).\ MVL(ip) =$
(if $\exists a \in A$ s.t. $ip \in a$ then $n+1$ else 1)

Condition 1 is met if Proposition 1 holds when using this algorithm, since no attack of A may occur in fewer than $n+1$ faults, and any occurence of fewer than $n+1$ attacks are detected, according to condition 2 of vulnerability level definition. In the same way no new attack is introduced because, under $n+1$, they are detected (condition 2).

2) Minimal attack-based placement algorithm: Multi-fault analysis often results in redundant attacks: for instance if we have a path being a 1-fault attack on an injection point ip, all attacks with the same prefix until ip are considered as redundant. We can compute the set of minimal attacks [25] as explained in definition 6. Considering at first minimal attacks happens to be useful for larger examples [18].

Definition 6 (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 an unique attack a in $Minimal(E)$ such that the formed by the sequence of $faults(a)$ is a *proper prefix* of the word $faults(b)$.

A substantial improvement of the previous algorithm is to protect all the IPs of the set *Minimal(A)* at level $n+1$. Indeed, as a direct consequence of Definition 6, to each *redundant* (i.e. non minimal) attack b in A corresponds a minimal attack a which is a *proper prefix* of b. Hence, protecting all the IPs of a is sufficient to prevent b. This new version is called `min` in Table V.

D. Block placement algorithm

A way to further improve the previous algorithms is to **reduce** the set of IPs to protect at level $n+1$ per minimal attack of A. Several heuristic combinations can be used to choose this subset. The placement algorithm proposed in Algorithm 2 – called `block` in Table V – proceeds as follows: each minimal attack $a \in A$ is traversed by increasing number of faults, followed by a decreasing number of redundant attacks associated to a. Then, for each attack which is not already protected, the injection point with the highest occurrence in the set A is selected.

Algorithm 2 Block placement algorithm

Require: a program P, a set of attack paths A of P up to a n faults
  $Protected \leftarrow \emptyset$ ▷ set of protected paths of A
  $IpProtected \leftarrow \emptyset$ ▷ set of protected ips of P
  $MVL \leftarrow \emptyset$ ▷ map of required vulnerability level for each IP of P
  for ip in $IPs(P)$ **do** ▷ Init MVL to 1
    $MVL[ip] \leftarrow 1$
  end for
  for k in 1 to n **do**
    $A_k \leftarrow$ attacks of A with k faults
    for a in A_k **do**
      if $IPs(a) \cap IpProtected \neq \emptyset$ **then**
        $Protected \leftarrow Protected \cup \{a\}$
      else ▷ If a is not already protected
        choose $ip \in IP$ with a maximal number of occurrences in $A \setminus Protected$
        $MVL[ip] = n+1$
        $IpProtected \leftarrow IpProtected \cup \{ip\}$
        $Protected \leftarrow Protected \cup \{a\}$
      end if
    end for
  end for
  return MVL

Proposition 1 always holds because of the complete exploration of attacks set A for the program P and the fault models

[8]We only have to guarantee a k-completeness, including the expected combination of faults.

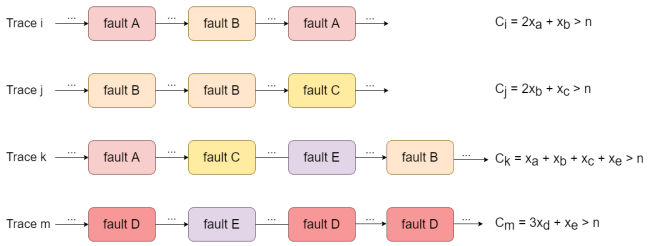

Fig. 4. Constraints on vulnerability levels given by traces in A

M. The set of attacks A indicates which unprotected IP invalid behavior (i.e. faulted) can lead to a violation of S, and isolation analysis guarantees that an IP protected with $vl = n+1$ cannot produce any abnormal behavior in n faults. As at least one IP per attack in A is protected with $vl = n + 1$ with the *block* algorithm, the composition of the generation of the attacks A and isolation analysis allow to guarantee that Proposition 1 holds.

E. Optimal distributed placement algorithm

Finally, the last algorithm we propose aims to **distribute** protections among the IPs to ensure that the "global" vulnerability level of P' for each attack of A is at least $n + 1$. This algorithm – called `opt` in Table V – is illustrated on Figure 4, by associating to each attack a a *constraint* C_a telling that the **sum** of the vulnerability levels of each IP of a, should be strictly greater than n. More precisely, to prevent (up to n faults) an attack a of A with IPs $[ip^a_1, \dots ip^a_k]$, the constraints to fulfill are expressed by the equality C_a of the form $\alpha_1 x_1 + \cdots + \alpha_k x_k > n$, where, for each injection point ip^a_i, x_i is the minimal vulnerability level required to protect ip^a_i and α_i is the number of occurrences of ip^a_i in a. The solutions $\{x^{\#}_i\}$ obtained for this *Integer Linear Programming* (ILP) problem, minimizing the objective function $x_1 + x_2 + ... + x_m$, are then the **optimal** vulnerability levels required to protect P.

This distributed placement algorithm prevents existing P attacks of A (Condition 1 of Proposition 1). Indeed, if we consider an l-faults attack $a = [ip^a_1, ip^a_2, \cdots, ip^a_l]$ of P, then executing the same attack in P' would now require at least $n+1$ faults since constraint C_a is satisfied. Condition 1 holds if the non-nominal behaviors produced by partially protected IP (IP protected with $vl < n + 1$) is subsumed by the fault models considered when building A from P. It is the case in our fault models and countermeasures, but for instance, for a *set-to-0* fault model, if the protected scheme can produce other integer values besides its vulnerability level, then Condition 1 does not hold and another more general fault model have to be used when generating A.

F. Discussion

From a practical point of view, some of the hypotheses we considered so far to produce a program P' robust up to n faults may not always hold. We discuss below which guarantees we can still obtain in these situations.

1) Non representative attack set: Computing a representative attack set A may not always be possible, in particular for large programs and large values of n. However, in such a situation, the last four algorithms we propose still ensure that P' **is more robust than** P in the sense of Definition 3.

2) Incomplete catalog: Another issue may arise when the catalog C is not complete with respect to the mapping MVL, namely some required protection schemes are not available. Here again, useful information regarding the robustness of P' can still be provided to the user, for instance:

- the subset U of **unprotected attacks**, that is the ones in A containing some unprotected enough injection points in P' ;
- the **robustness level** obtained for P', that is the smallest number of fault required to achieve an attack in U, taking into account applied vulnerability level to IPs.

VI. Experimentations

In this section, we present the results obtained with the placement algorithms introduced in last section. We consider the test inversion and data load mutation fault models, with their combination, and the Test i-plication and Load i-plication countermeasures. We used a set of programs from the FISSC collection [12], three are presented here:

- *VP* a version of the verifyPIN program corresponding to a generalization of our introductive examples (section II),
- *FU* an implementation of a firmware updater (using a systematic test duplication allowing to evaluate how the placement algorithm behave if P already contains some countermeasures),
- and finally *MCMPS* that corresponds to a secure comparison of arrays, using multiple calls to the standard `memcmps` function and masks.

All these programs are already be used for large experimentations [6] and will be made freely available. Our results are summarized in Table VI. Columns "PM" gives the program P to protect and the fault models m to consider. Column "IPs" gives the total number of injection points in P wrt m. Five placement algorithms are considered, indicated in column "Algorithm": `naive`, `atk`, `min`, `block` and `opt`. Column "#added-cms" indicates the total sum of vulnerability levels required for P', for each algorithm, up to $n = 4$ faults. In those experimentation, the catalog is complete for all fault models, test i-plication and load i-plication giving a protection scheme for any required vulnerability level.

As expected, distributed approach *opt* gives at least better results than `block` algorithm that is at least better than systematic approach. In particular the `naive` one, used by all tools adding countermeasures in a systematic way (disregarding actual attacks) is clearly outperformed. More generally, the sum of protection generated by algorithms follows the order $opt \leq block \leq min \leq atk \leq naive$. In case of unique fault, all algorithms using the set of attacks (starting from `atk`) gives the same results because only one IP can be protected per attacks and thus require a vulnerability level of $n + 1 = 2$ to ensure robustness.

TABLE VI
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		`atk`	3	8	12	16
		`min`	3	8	12	16
		`block`	3	6	9	12
		`opt`	3	6	9	12
MCMPS	12	`naive`	12	24	36	48
with test inversion		`atk`	0	0	0	16
		`min`	0	0	0	16
		`block`	0	0	0	4
		`opt`	0	0	0	1
MCMPS	15	`naive`	15	30	45	60
with data load mutation		`atk`	1	6	15	32
		`min`	1	6	15	32
		`block`	1	4	6	8
		`opt`	1	3	5	7
MCMPS	27	`naive`	27	54	81	108
with test inversion		`atk`	1	8	24	56
+ data load mutation		`min`	1	8	24	56
		`block`	1	6	9	12
		`opt`	1	3	5	8
FU	42	`naive`	42	84	126	168
with test inversion		`atk`	0	28	42	88
		`min`	0	28	42	72
		`block`	0	14	21	28
		`opt`	0	7	14	21
FU	2	`naive`	2	4	6	8
with data load mutation		`atk`	1	4	6	8
		`min`	1	2	3	4
		`block`	1	2	3	4
		`opt`	1	2	3	4
FU	44	`naive`	44	88	132	176
with test inversion		`atk`	1	32	60	96
+ data load mutation		`min`	1	32	60	80
		`block`	1	16	24	32
		`opt`	1	9	17	25

VP shows optimal results with `block` placement algorithm, meaning that distribution of protection is not required to achieve optimal placement.

For MCMPS program, results are presented for both fault models and their combination. As no test inversion attacks are possible below 4 faults, only `naive` placement requires countermeasure in less than 4 faults. The 4-faults attack with test inversion corresponds to the attack of the same IP four times, allowing the `opt` algorithm to protect this IP only with $vl = 2$. Only one 1-fault attack is possible with data load mutation, giving the same results for every algorithm (except `naive`). For both fault models, distributed placement is required to obtain optimal protection sum.

Similarly, FU results are presented with both fault models and combined fault model. Distributed placement is also required to achieve optimal placement for test inversion, and thus for the combination of fault models.

For both MCMPS and FU, the protection applied for combined fault model is not always the sum of the protection (except for 1-fault). However, the protection for the combination can never be smaller than the sum of the protection for each fault model separately.

Table VII resumes the properties of the five placement algorithms (column "Algo.") presented in this paper. Column "Type" indicates if the algorithms use systematic, block or distributed placement approach. The column "Guarantees P'" precises if the algorithm is robust (with a complete A attacks set and a complete catalog), and if the algorithm is guaranteed to find an optimal solution. Finally, the "Required analysis" corresponds to the analysis that are necessary for the algorithm: attack analysis ("AA"), redundancy analysis[9] ("Red") and hotspots analysis ("HS").

TABLE VII
CHARACTERISTICS OF THE PLACEMENT ALGORITHMS

Algo.	Type	Guarantees P' Robust	Optimal	Complexity	Required analysis AA	Red	HS
naive	syst.	✓	-	$O(t)$	-	-	-
atk	syst.	✓	-	$O(t)$	✓	-	-
min	syst.	✓	-	$O(t)$	✓	✓	-
block	block	✓	-	$O(t)$	✓	✓	✓
opt	distr.	✓	✓	NP-Complete	✓	✓	-

Performance wise, DSE remains the limiting factor of the methodology. Redundancy analysis can approach the computation time of KLEE in some example such as FU, but attacks analysis and hotspots analysis are $O(t)$ (with t the set of attack traces). The ILP solving worst case is $NP - complete$, but our experimentation shows that its computation time is very low. Indeed, low order attacks add strong constraints to the ILP and if 1-fault attacks exist, the distributed algorithm has to protect those IPs with $vl > n$.

VII. CONCLUSION

In this paper, we proposed an innovative methodology to assist developers when adding countermeasures to protect a program against *multi-fault attacks*, nowadays considered at the state of the art by certification authorities. Generalizing the existing single fault hardening approaches is difficult due to the attack surface introduced by countermeasures, potentially adding new attacks and new paths to explore. To handle this inherent complexity, we propose a *compositional* approach combining *isolation analysis*, allowing to compute the vulnerability level of a protection scheme characterizing how many faults are required to violate the postcondition of an IP, and *placement algorithms*, relying on a representative set of attacks to select which minimal vulnerability level should be applied on each IP in order to obtain robustness in n faults.

The *isolation analysis* allows to reason about a protection scheme locally. The adequacy of the scheme allows to verify if the protection actually blocks the single fault attacks and the *vulnerability level* allows to abstract the additional behaviors of the protected scheme introduced by the countermeasures. The key of this analysis is the specification of the nominal behavior of instructions, how it can be impacted by faults and how deviations can be detected.

Formally establishing the robustness of a countermeasure scheme against an attacker model is not new. In the context of CFI many protections have been proposed and proved, as in [3]. In the context of fault injection, formal methods have

[9]Redundancy analysis corresponds to the computation of the minimal attacks.

been used to establish the effectiveness of countermeasures [15], [16], [20]. But these works are dedicated to particular forms of countermeasures and specific fault models, and they address single faults only. Here we propose a generic approach extended to multi-fault. To the best of our knowledge, no analysis has been already proposed to take into consideration the attack surface of a countermeasure, because this question is not relevant single fault.

Five placement algorithms have been presented in this paper. Under some reasonable hypothesis these algorithms guarantee the robustness of the protected program up to n faults, w.r.t. the considered fault model and attack objective. In particular, we assume that a representative attacks set has to be computed beforehand, similarly to what is done in simulation based approaches, when the program is protected with respect to weaknesses revealed by profiling techniques.

This methodology has been experimented with (combinations of) test inversion and data load mutation fault models on a set of program from the FISSC collection [1]. These experiments demonstrate that our approach is effective on realistic codes, and allows to drastically reduce the cost of protections to insert w.r.t. systematic strategies consisting in protecting all IPs.

Experimentation focused on test inversion and data mutation fault models which are standard fault models at software level [18]. Test i-plication and load i-plication are protections scheme that can provide any vulnerability level required for those fault models. Our approach could be extended to other countermeasures that propagate an internal state, like the one proposed in [10], requiring then to take this internal state into account during isolation analysis. However, some limitations may occur with more "distributed" countermeasures, namely when the protection scheme consists in modifying the target code "as a whole", at several locations. This could be hardly compatible with the local isolation analysis we propose.

Similarly, fault models based on (arbitrary) jumps would require to take into account all possible entry points and output points of a protection scheme in order to run isolation analysis. In particular, if a fault can produce a jump anywhere outside the protected scheme, the postcondition to be verified in order to correctly compute its vulnerability level may be difficult to specify. More generally, this raises the question of knowing which fault models can be protected by our approach (with IP granularity protection), require a more global transformations to be robust, or cannot be make robust at all [14].

In another field, our methodology can be applied to fault tolerance analysis, using a bit-flip data model and load duplication, for instance. However, some crypto-based code (with high cyclomatic complexity) are difficult to explore with Lazart that forks the DSE exploration each time a fault can be injected. If another tool or method produces the set of attacks A, our compositional approach can be applied to make the program robust. In the same way, side-channel attacks evaluation would require to handle this type of attack objective for the generation of A.

Finally, this work has been implemented using LLVM level and a future work could be to implement it on a lower level. Nevertheless, this would require to adapt isolation analysis to those specific fault models and sensitive schemes.

ACKNOWLEDGMENT

This work is supported by the ARSENE and SECUREVAL projects funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference ANR-22-PECY-0004 and ANR-22-PECY-0005.

REFERENCES

[1] "FISSC: the Fault Injection and Simulation Secure Collection," https://lazart.gricad-pages.univ-grenoble-alpes.fr/fissc.

[2] "Application of Attack Potential to Smartcards and Similar Devices," Joint Interpretation Library, Tech. Rep., June 2020.

[3] 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.

[4] 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*.

[5] 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.

[6] E. Boespflug, C. Ene, L. Mounier, and M.-L. Potet, "Countermeasures Optimization in Multiple Fault-Injection Context," in *FDTC"*, Sep. 2020.

[7] 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*.

[8] 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.

[9] C. M. Committee. (2012, Sep.) Common Criteria for Information Technology Security Evaluation. [Online]. Available: http://www.commoncriteriaportal.org/files/ccfiles /CCPART3V3.1R4.pdf

[10] F. de Ferrière, "A compiler approach to cyber-security," 2019 European LLVM developers' meeting, 2019. [Online]. Available: https://llvm.org/devmtg/2019-04

[11] S. Delarea and Y. Oren, "Practical, low-cost fault injection attacks on personal smart devices," *Applied Sciences*, vol. 12, no. 1, 2022.

[12] 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*.

[13] 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.

[14] T. Given-Wilson, N. Jafri, J.-L. Lanet, and A. Legay, "An automated formal process for detecting fault injection vulnerabilities in binaries and case study on present," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 293–300.

[15] 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*, Nov. 2015.

[16] K. Heydemann, J. 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.

[17] C. Hillebold, "Compiler-assisted integrity against fault injection attacks," Master's thesis, University of Technology, Graz, December 2014.

[18] G. Lacombe, D. Féliot, 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*, pp. 1–18, 2023.

[19] J.-F. Lalande, K. Heydemann, and P. Berthomé, "Software countermeasures for control flow integrity of smart card C codes," in *ESORICS - 19th European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, vol. 8713. Springer International Publishing.

[20] 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.

[21] 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, 2014*.

[22] 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.

[23] S. Nashimoto, N. Homma, Y.-i. Hayashi, J. Takahashi, H. Fuji, and T. Aoki, "Buffer overflow attack with multiple fault injection and a proven countermeasure," *Journal of Cryptographic Engineering*, vol. 7, no. 1, pp. 35–46, 7 2017.

[24] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[25] 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.

[26] 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.

[27] 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 Conference on Computer and Communications Security*. ACM, 2019.

[28] 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.

[29] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "Swift: software implemented fault tolerance," 04 2005, pp. 243– 254.

[30] 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.

[31] N. Theissing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *DATE Conference*. IEEE, 2013.

[32] N. Timmers and A. Spruyt, "Bypassing secure boot using fault injection," in *Black hat Europe 2016*.

[33] J. Van den Herrewegen, D. Oswald, F. D. Garcia, and Q. Temeiza, "Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 1, p. 56–81, Dec. 2020.