Countermeasures Optimization in Multiple Fault-Injection Context

Etienne Boespflug
Cristian Ene
VERIMAG
University of Grenoble Alpes (UGA)
Grenoble, France
Email: Firstname.Lastname@univ-grenoble-alpes.fr

Laurent Mounier
Marie-Laure Potet
VERIMAG
University of Grenoble Alpes (UGA)
Grenoble, France
Email: Firstname.Lastname@univ-grenoble-alpes.fr

***Abstract*—Fault attacks consist in changing the program behavior by injecting faults at run-time, either at hardware or at software level. Their goal is to change the correct progress of the algorithm and hence, either to allow gaining some privilege access or to allow retrieving some secret information based on an analysis of the deviation of the corrupted behavior with respect to the original one. Countermeasures have been proposed to protect embedded systems by adding spatial, temporal or information redundancy at hardware or software level. First we define Countermeasures Check Point (CCP) and `CCPs`-based countermeasures as an important subclass of countermeasures. Then we propose a methodology to generate an optimal protection scheme for `CCPs`-based countermeasure. Finally we evaluate our work on a benchmark of code examples with respect to several Control Flow Integrity (CFI) oriented existing protection schemes.**

***Keywords*-multiple fault-injection; code analysis; countermeasure optimization; 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 essentially 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 [1], [2], or by exploiting some weaknesses in the power management modules [3], [4], [5], [6].

As a result, developers have to integrate mitigation mechanisms into their product as dedicated hardware and/or software *countermeasures* whose purpose is to detect (or even prevent) runtime security violations. In addition, well established *certification schemes* [7], [8] have been developed to validate the protection level achieved. In practice, the process of designing robust applications mainly consists in identifying assets to be protected, protect them and finally check their robustness with respect to state-of-the art attacks. In particular, today's applications must now be protected against (spacial or temporal) *multi-fault injection* [9], [10], namely when several faults can be injected at various successive locations during an execution, making the evaluation and development processes very challenging. Due to the growing complexity of these tasks, designing (and certifying) secure applications requires specific tools to evaluate their robustness [11], [12], even under single-fault assumptions.

However, countermeasure integration still raises very challenging issues from the developer point of view, namely: how to choose and set up the most appropriate protection scheme, possibly with respect to some *application dependent* security properties, and with specific security/performance trade-offs in mind? More specifically, how to identify which parts of the code should *really* be protected, how to avoid useless runtime checks, and redundant countermeasure trigger points? Note that all these questions become even more crucial in a multi-fault context, where countermeasures themselves can be attacked, and where a "try and test" approach becomes no longer effective. To the best of our knowledge, there is no current work addressing such countermeasure analysis, as an intrinsic problem. The objective of this paper is precisely to address this issue and to assist the developer in the countermeasure insertion process. In particular, we provide the following contributions:

1) we formulate the problem of *test-based countermeasure optimization*, where a program countermeasure is a set of trigger points guarded by side-effect free boolean conditions;
2) we propose a methodology to generate an optimal protection scheme for such application embedded countermeasures, in a multi-fault injection context;
3) we provide an implementation of this approach, based on the Lazart tool [13];
4) finally, we evaluate our work on a benchmark of code examples with respect to several existing protection schemes targeting CFI.

Section II introduces the context of multiple fault-injection through a motivating example. Section III defines our notion of so-called `CCPs` -based countermeasure and discusses about common issues related to countermeasure

978-1-7281-9562-9/20/$31.00 ©2020 IEEE
DOI 10.1109/FDTC51366.2020.00011

analysis and comparison. Section IV defines the four steps of our methodology, and Section V shows the results obtained with our implementation on the benchmark we considered. Section VI presents some related work. Finally, we conclude with Section VII, discussing some limitations of our solution and directions for future work.

II. Fault injection attacks

A. Motivating example

Listing 1 is an excerpt of a *verifyPIN* program taken from the FISSC public benchmark [14]. Function `verifyPIN` checks that an user-supplied `userPIN` corresponds to a (secret) `cardPin` in order to grant the authentication. `g_ptc` is a counter checking that the maximal number of consecutive unsuccessful tries was not exceeded.

Listing 1. Example: a PIN code verifier

```
1  BOOL byteArrayCompare(UBYTE* a1,
2          UBYTE* a2, UBYTE size) {
3    BOOL result=TRUE;
4    UBYTE i;
5    for(i=0; i < size; i++)
6      if(a1[i] != a2[i]) result=FALSE;
7    if(i != size) killcard();
8    return result;
9  }
10
11 BOOL verifyPIN() {
12   if(g_ptc > 0) {
13     if(byteArrayCompare
14      (userPin, cardPin, PIN_SIZE)){
15        authenticated=1; // Auth
16       g_ptc=3;
17       return TRUE;
18     } else {
19        g_ptc--;
20        return FALSE;
21     }
22   }
23   return FALSE;
24 }
```

An expected *security property* Φ is that an user get authenticated (line 14) `only if` the correct `userPin` was supplied. Φ can be strengthened to a property Φ' in order to prevent fault injections that increase the value of the counter `g_ptc`. The property Φ clearly holds without any fault injection. However, when considering the (classical) *test inversion* fault model i.e., assuming that each control-flow condition can be "negated" at runtime, this property is of course no longer ensured. In particular an attacker may enforce execution of line 14 by changing the result of `byteArrayCompare()` (line 12), or may perform a brute force attack by repeatedly evaluating as true `g_ptc>0` (line 11). Note that function `byteArrayCompare()` is protected against an incorrect number of iterations of its internal loop (line 6), which is an example of counter-measure. The number of (distinct) successful test inversion attack scenarios against Φ provided by the tool Lazart is summarized in Table I[1].

Table I
Attacks on verifyPIN with test inversion

Fault bound	0 faults	1 faults	2 faults	3 faults	4 faults
Attacks	0	1	6	10	12

A 2-faults attack, using the test inversion model, consists in inverting the first loop condition in `byteArrayCompare` (line 5 of listing II) and then the loop counter verification (line 7) to be authenticated.

B. Faults, fault models and attacks

We assume in the following that a program *execution trace* is modelled in a standard way as a sequence of transitions "$\rightarrow$" between *configurations*, where a configuration c is a pair (l, μ) with l being a program control location and μ a memory content. In this paper we are going to consider so-called *symbolic* execution traces, where the memory content μ is a *symbolic store*, mapping program variables either to concrete values or to symbolic expressions that can contain symbolic meta-variables, in addition with a path predicate, as used in Dynamic Symbolic Execution (DSE) [15]. For a program p and a configuration c, we then denote by $Exec(p, c)$ the set of such symbolic execution traces of p, starting from configuration c.[2].

We define a **volatile fault** as a modification of an instruction or of a memory location occurring at some injection point, **during the program execution** (i.e., the program itself is not modified). A *fault model* f is then characterized by a set of *injection points* (the program configurations where faults of f may apply) and the "effect" of these faults on the program execution (how the memory is modified, which new configuration is reached, etc.). A fault model f will be formalized by a labeled transition relation between program configurations, noted $c \hookrightarrow c'$, defining the configuration c' reached when injecting a fault at configuration c. Note that this transition relation is defined only if c is an injection point for the fault model f. As an example, for the test inversion fault model, injection points are conditional branch instructions and transition relation $\hookrightarrow$ allows to reach the configuration corresponding to the incorrect branch target (according to the original instruction semantics).

A *n-faults attack* on a program p, noted a_p^n, is then defined as an execution trace containing sub-sequences of "nominal" execution steps σ_i (i.e., following the program semantics) interleaved with n *faults* $\hookrightarrow$, injected between the σ_i according to the considered fault models. More formally, for a program p with initial configuration c_0:

$$a_p^n =_{def} [(\sigma_0, \hookrightarrow_0), (\sigma_1, \hookrightarrow_1), \ldots, (\sigma_{n-1}, \hookrightarrow_{n-1}), \sigma_n]$$

where

$$\sigma_i = [c_i \rightarrow c_i^1 \rightarrow \ldots \rightarrow c_i^{k_i}] \in Exec(p, c_i) \ \wedge \ c_i^{k_i} \hookrightarrow_i c_{i+1}$$

[1] we consider here *symbolic* attack scenarios, see section II-B

[2] or simply $Exec(p)$ when c is the initial program configuration

Note that this definition allows a same injection point to occur several times in an attack (with or without fault injections), and allows attacks combining several fault models. Moreover, we distinguish between *successful* attacks, able to violate some desired security property Φ of the target program, and *unsuccessful* ones.

III. Fault injection countermeasures

Intuitively, a software *countermeasure* is a code transformation which preserves the original program behaviour under nominal execution conditions and which aims to enforce some security properties in case of fault injection. Usually, countermeasures are designed with respect to some precise fault model. Beyond making the software more secure, they may also take into account other objectives like performance, automation easiness, etc.

A. CCP-based Countermeasures

We refine the concept of *test-based countermeasures* with the *CCP-based countermeasures*[3], which are divided in two parts:

- The *Countermeasure Check Points* (CCPs), are control points in the program where one checks if some safety condition b about the program state holds. They correspond to a fault detection spot in the program. We assume that each CCP has associated an unique identifier i.
- The *Countermeasure Structures* (CS) concern the pieces of code (changes w.r.t. the original code or newly introduced) that correspond to computations that are useful to CCPs.

In practice, when triggered, a CCP can lead to a program shutdown or restart, some error reporting etc. In the following examples we will denote that using a special function `killcard()`.

In order to model CCPs in our framework, we assume at the semantic level that there is a special command **trigger**(b, i) that allows to the CCP i to check if the Boolean condition b is respected by the current configuration. Formally, we extend the operational semantics from subsection II-B with the two additional rules below (assuming that l' is the successor location of l):

$$\frac{p(l) = \mathbf{trigger}(b,i) \qquad \mu(b) = false}{(l,\mu) \overset{\mathsf{t}(i)}{\rightarrow} (l',\mu)} \tag{1}$$

$$\frac{p(l) = \mathbf{trigger}(b,i) \qquad \mu(b) = true}{(l,\mu) \rightarrow (l',\mu)} \tag{2}$$

Rule (1) corresponds to the case where condition b does not hold, and hence an *alarm* needs to be raised at control point l, whereas rule (2) corresponds to the case where the program execution can continue normally. Note that we assume that the command **trigger**(b, i) is side-effect free, that is, it does not change the memory content μ, in the case that the condition b is not satisfied, it only provokes a labelled transition. Of course, this side-effect free hypothesis may be tuned depending on the code granularity level we consider, the main assumption being that at some point there exists software mechanism able to trigger a countermeasure in an atomic way.

For an attack a_p^n, we denote by $TSeq(a_p^n)$ the underlying sequence of triggered alarms, i.e. $TSeq(a_p^n)$ is defined as the sequence obtained from a_p^n by erasing all the transitions that are not of the form $\overset{\mathsf{t}(i)}{\rightarrow}$,

$$TSeq(a_p^n) = [c_{i_0} \overset{\mathsf{t}(i_0)}{\rightarrow} c_{i_0+1}, \ldots c_{i_{n-1}} \overset{\mathsf{t}(i_{n-1})}{\rightarrow} c_{i_{n-1}+1}]$$

Definition 1: An n-faults (successful) attack a_p^n ($n \geq 1$) is called **detected** iff $TSeq(a_p^n)$ is non-empty, i.e. at least one alarm was triggered. If $TSeq(a_p^n)$ is empty, then a_p^n is called **undetected**.

B. Test duplication

In this section, we introduce our methodology using a simple countermeasure example, `Test duplication`. This countermeasure transforms each control-flow conditional branching by introducing two CCPs, one for the true branch and one for the false one. We choose here to compute the condition value only once, i.e., its result is reused in each of the two CCPs.

The Listings 2 (respectively 3) show the general C-level transformations applied to each if statement (and respectively to each while loop). Assuming that the condition `c1` is side-effect free, the statement "`if (c1) killcard();`" will be modelled in our abstract semantics as a command **trigger**(c1, i) where **i** is a fresh CCP identifier. Even if contrary to the C-semantics of the function "`killcard();`", the command **trigger**(c1, i) does not interrupt the program, remember that we are interested in the detection of fault injection attacks and this modeling preserves this property.

The Listing 5 shows the resulting program obtained after applying the `Test duplication` countermeasure to the PIN code verifier example from Listing 1, with the CCPs numbered in comment.

Listing 2. Test duplication on if statements

```
1  // Source
2  if(condition)
3    do_something();
4
5  // Protected
6  BOOL c1;
7  if(c1 = condition)
8  {
9    if(!c1) killcard();
10   do_something();
11 }
12 else if (c1) killcard();
```

[3]Not all countermeasures can be modelled in this way.

Listing 3. Test duplication on while loops

```
1  // Source
2  while(condition)
3    do_something();
4
5  // Protected
6  BOOL c1;
7  while(c1 = condition)
8  {
9    if(!c1) killcard();
10   do_something();
11 }
12 if(c1) killcard();
```

IV. Optimizing a CCP-based countermeasure

Our goal is to simplify a CCP-based countermeasure without sacrificing its security wrt a fault model and a security property. Concretely, our approach is exploring the set of all (detected and successful) attacks and their underlying sequences of triggered alarms and try to find all CCPs that can be safely removed without increasing the set of undetected successful attacks.

The methodology takes as parameter the attack model: a *fault model* and a *security property*. It takes a program P in input and returns a program P'' as output. It consists in the following successive steps:

1) Generate the instrumented program P' from the program P containing CCPs to analyze.
2) Generate a list of (symbolic) attacks for P, for instance using a Dynamic-Symbolic Execution.
3) Compute the *CCP Classification*.
4) Choose a *removal strategy* and use *CCP selection algorithm* to find the optimal sets of CCPs to be removed.
5) Use *structure removal rules* to remove countermeasure's structures (variables, parameters...) corresponding to the set of removed CCPs. We denote by P'' the program obtained after this step.

P'' is the optimized protected version of P. P'' should not contain additional undetected attacks compared to P'.

A. Instrumentation and CCP requirements

The instrumentation step of our methodology takes as input the original program P strengthened with a CCP-based countermeasure, and produces the instrumented program P' obtained by replacing the `killcard()`-like alarm handlers by a *Lazart specific function* (LZ_ALARM) allowing to register all run-time triggered alarms. This transformation implements the semantics of the **trigger** command introduced in Section III-A. An example of such replacing is shown by the Listing 4, where i is a fresh integer identifier.

Listing 4. Instrumentation of a killcard CCP

```
1  // Original code with CCP-based countermeasures
2  if(condition)
3      killcard();
4
5  // Resulting code after instrumentation
6  if(condition)
7      LZ_ALARM(i);
```

Table II
CCP CLASSIFICATION ON VERIFYPIN STRENGTHENED WITH THE TEST DUPLICATION COUNTERMEASURE WITH UP TO 4 TEST INVERSIONS

CCP	0	1	2	3	4	5	6	7	8	9	10
1 fault	I	I	I	R	I	I	I	I	N	I	R
2 faults	R	I	R	R	R	R	R	R	R	I	N
3 faults	N	I	N	N	N	N	R	N	N	I	N
4 faults	N	I	N	N	N	N	R	N	N	I	N

B. CCP Classification

The classification algorithm takes as input the set $\mathcal{A}$ of attacks and it partitions the set $\mathcal{C}$ of CCPs in three classes: **Inactive, Necessary or Repetitive**. We associate with each attack a a *repetition level* $\mathcal{R}(a)$ as being the number of **different** CCPs triggered by a.

The classification algorithm focuses on the minimum repetition level $L_m[C_i]$ of each CCP C_i on the set of (symbolic) attacks traces which are:

- successful: validate the oracle.
- blocked: at least one CCP is triggered in the trace.

Each CCP C_i is classified depending on its minimal repetition level among all the considered traces:

- Inactive, if $\mathcal{L}_m(C_i) = \infty$ (never triggered);
- Necessary, if $\mathcal{L}_m(C_i) = 1$;
- Repetitive, if $1 < \mathcal{L}_m(C_i) < \infty$.

It is safe to remove the *Inactive* CCPs since they are never triggered. In the case of *Repetitive* CCPs, we apply an additional step, the CCP Selection Algorithm (see section IV-C) in order to compute an optimal set of CCPs that could be removed without adding new undetected attacks.

The Table II presents the CCP classification for the verifyPIN example strengthened with the test duplication countermeasure and up to 4 test inversion faults injected. One can remark that the example has 11 CCPs, and in the case of 2 test inversions, among these CCPs, only the last one is Necessary, two of them are Inactive and can be safely removed, and the remaining 8 CCPs are Repetitive.

C. CCP Selection

This step aims to compute an optimal set of Repetitive CCPs to keep with respect to the CCP classification. We assume that the Inactive CCPs were already removed and we also ignore the attacks that trigger at least a Necessary CCP. Hence the Selection Algorithm takes as input the set $\mathcal{C}_r \subseteq \mathcal{C}$ of Repetitive CCPs and the subset $\mathcal{A}_r \subseteq \mathcal{A}$ of attacks containing only Repetitive CCPs. It computes a minimal subset of Repetitive CCPs that guarantees that no new undetected attacks are added. The algorithm can be parameterized by a *cost or weight* function $\mathcal{W} : \mathcal{C}_r \to \mathbb{R}^+$ associating a cost $\mathcal{W}(i)$ to each Repetitive CCP i depending

on some desired property of the countermeasure (performance, code size etc). This function is lifted to sets R_i of CCPs in the usual way $\mathcal{W}(R_i) = \sum_{i \in R_i} \mathcal{W}(i)$. Then the *CCP Selection* corresponds to find the valid sets R_i with the minimal cost value $\mathcal{W}_{R_i}$, where a set of CCPs $R_i \subseteq \mathcal{C}_r$ is called **valid** if for each attack $a \in \mathcal{A}_r$, at least one CCP in R_i is triggered.

D. CS Removal

Once the set of removable CCPs has been computed, then the corresponding conditional tests can be removed without weakening the security. The next step is to remove unused countermeasure's structures as well. This step corresponds to useless code elimination, which is a common problem in code analysis. It can be done with a compiler's optimizer or static analysis tools (as Frama-C [16] for example).

The Listing 5 presents one optimal solution provided by our methodology where unnecessary CPPs and consequently, unused CS has been removed. The cost function used to produce these results is the constant function associating 1 to each CCP. The pieces of code depicted in green are kept, and the pieces of code depicted in red can be safely removed without introducing new undetected attacks.

Listing 5. Removed CCPs and CS are depicted in red (attacks in 2 faults)

```
1  BOOL byteArrayCompare(UBYTE* a1,
2           UBYTE* a2, UBYTE size) {
3       BOOL result = TRUE;
4       UBYTE i;
5       BOOL c_1 = FALSE;
6
7       for(i = 0; BOOL c_1 = i < size;
    i++) {
8           if(!c_1) killcard(); // CCP 2
9           if(BOOL c_2 = a1[i] != a2[i]) {
10              if(!c_2) killcard(); // CCP 4
11              result = FALSE;
12          } else
13              if(c_2) killcard(); // CCP 5
14      }
15      if(c_1) killcard(); // CCP 3
16      if(BOOL c_3 = i != size) {
17          if(!c_3) killcard(); // CCP 6
18          killcard(); // CCP 11
19      } else
20          if(c_3) killcard(); // CCP 7
21      return result;
22  }
23
24  BOOL verifyPIN() {
25      if(BOOL c_1 = g_ptc > 0) {
26          if(!c_1) killcard(); // CCP 0
27          if(BOOL c_2 =
    byteArrayCompare(g_userPin,
     g_cardPin, PIN_SIZE) == TRUE) {
28              if(!c_2) killcard(); // CCP 8
29              g_authenticated = 1;
30              g_ptc = 3;
31              return TRUE;
32          } else {
33              if(c_2) killcard(); // CCP 9
34              g_ptc--;
35              return FALSE;
36          }
37      } else
38          if(c_1) killcard(); // CCP 1
39      return FALSE;
40  }
```

V. Experimentation

In this section, we present and discuss several experiments involving three generic countermeasures. These experiments were performed using the tool Lazart. We start by briefly presenting this tool, then we describe the examples we used, together with a set of countermeasures we considered. We conclude the section by an analysis of the obtained results.

A. Lazart

Lazart [13] is a tool allowing to check the robustness of a software under multi-fault injections. It takes as input an LLVM Intermediate Representation (IR), a fault model, and a security property. It relies on a 2-steps approach:

- First, a *high order mutant* is generated from the initial LLVM representation. This mutant statically encodes all the possible injected faults (as symbolic boolean values) according to the fault model. In this paper we consider the *test inversion* fault model.
- Then, a *dynamic-symbolic exploration*, performed by Klee [17], produces all the successful symbolic attacks with respect to the security property.

In this work, we first needed to extend Lazart with features specific to our approach (in order to model the CCPs and to take them into account in the symbolic execution traces). Then, we provide to the enhanced Lazart tool, as input an instrumented program as described in section IV-A and we get the set of all ordered sequences of triggered alarms corresponding to the successful symbolic attacks. This set is the input for the CCP Classification step.

B. Studied Examples

Our experimentation are performed on various programs from the FISSC [14] benchmark enriched with a custom *firmware updater* example. Each of these programs will give rise to several versions obtained by adding the countermeasures shown in the next section. These examples are briefly described below[4].

1) VerifyPIN (VP): this is the running example presented in the previous sections. Several security properties are considered (see Section V-D).

2) GetChallenge (GC): this program is an example of a nonce generation. The security property asserts that the nonce is updated with a randomly generated value. Note that the initial program already contains some CCP-based protections (shadow stack, loop counter, etc.) and can thus be directly analyzed with our approach.

[4]they are fully available at https://lazart.gricad-pages.univ-grenoble-alpes.fr/home/fdtc20

3) AES (AES): this program is an implementation of the standard AES encryption scheme. We also considered in isolation the specific `AddRoundKey` step (ARK).

4) Firmware Updater (FU): this program represents an updater for a firmware. The studied security property is the following: *the firmware is updated only upon request and none of its pages are corrupted and the correct loading address has been used.*

C. Studied countermeasures

To evaluate our approach we consider three oriented `CCP`-based countermeasures described below. All these countermeasures aim to enforce the Control-Flow Integrity (CFI) by detecting at runtime possible alterations of the expected control-flow[18].

1) Test Duplication (TD): this countermeasure has been introduced in Section III-B.

2) SecSwift ControlFlow (SSCF): this countermeasure [19] associates an unique identifier to each basic block (i.e., an atomic sequence of instructions) and uses a *xor*-based mechanism to ensure that the correct branch has been taken. The figure 1 shows how a conditional branching is protected by introducing two `CCPs` (the `secswift_assert` statement is equivalent to our abstract `alarm` command).

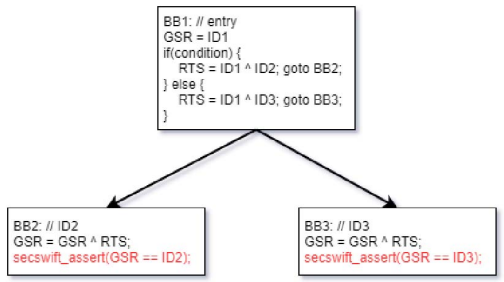

Figure 1. SSCF transformation scheme for an if statement

3) LHB [20]: this countermeasure introduces step counters to protect against C-level instruction skips. Each counter check corresponds to a `CCP` in our approach.

D. Results

Table III shows the results of the `CCP` optimization for several (program, countermeasure) pairs. Entries in column 1 are the pairs we consider, column 2 is the corresponding total number of `CCPs` and the remaining columns give the percentage of removed `CCPs` for up to 3 fault injections. Several remarks can be drawn from this table:

- the ratio of removed `CCPs` highly depends on the considered program and countermeasures;
- as expected, it decreases with respect to the number of injected faults;
- the high number of removed `CCPs` in the case of the LHB countermeasure (VP + LHB line) is due to the fact our fault model (Test Inversion) is weaker than the one targeted by this countermeasure.

Table III
PERCENTAGE OF REMOVED CCP FOR EACH EXPERIMENTATION

Program	CCP	1 fault	2 faults	3 faults
VP + TD	11	72%	63%	18%
VP + SSCF	13	92%	76%	23%
VP + LHB	31	93%	93%	32%
FU + TD	14	0%	0%	0%
FU + SSCF	24	12%	12%	8%
GC1	11	81%	72%	63%
GC1 + TD	39	37%	34%	34%
GC1 + SSCF	38	57%	28%	28%
AES RK + TD	2	50%	50%	0%
AES RK + SSCF	3	66%	33%	0%
AES C + TD	8	50%	50%	0%
AES C + SSCF	13	76%	61%	38%

Table IV
REMOVED CCP DEPENDING ON PROPERTY (VP + TD)

Property	1 fault	2 faults	3 faults
ϕ_{auth}	83%	72%	18%
ϕ_{ptc}	72%	63%	9%
$\phi_{auth} \vee ptc$	83%	72%	18%
$\phi_{auth} \wedge ptc$	72%	63%	9%
ϕ_{true}	18%	9%	9%

The percentage of removed `CCPs` also strongly depends on the considered security property. Table IV presents the results on *verifyPIN* using various security properties:

- ϕ_{auth}: being authenticated.
- ϕ_{ptc}: do not decrease the `g_ptc` counter in case of invalid pin code[5].
- $\phi_{auth} \vee ptc$.
- $\phi_{auth} \wedge ptc$.
- ϕ_{true}: consider all faulted executions (the security property being set to *true*).

For a same program a weaker security property increases the number of traces to consider, and thus reduces the number of CCPs that can be removed. The intuition behind this fact is that stronger security properties hold only on a small subset of the initial program traces, and hence this subset of traces is easier to protect with a reduced number of CCPs (with respect to the whole set of program traces, corresponding to the *true* security property).

Table V gives the execution times of our analysis [6] for 3 injected faults. Columns 2 and 3 present respectively the execution time and the number of completed paths of the DSE engine (Klee). The "traces" entry correspond to the number of symbolic attacks and the last column shows the duration of the `CCP` Optimization step (Classification + Selection). One can remark that the execution time of CCPO is very small compared to one of the DSE. Moreover, the Selection step is even much smaller compared to the

[5]in our example, and for Test Inversion, $\phi_{ptc} \Rightarrow \phi_{auth}$
[6]Performed on a laptop (Intel I7 2.6GHz, 16GB RAM)

Table V
TIME METRICS IN 3-FAULTS

Program	DSE (h)	Completed Paths	Traces	CCPO
VP + TD	0:00:03	7118	296	26ms
VP + SSCF	0:01:54	130 576	1005	89ms
VP + LL	0:38:24	1 173 312	37 347	371ms
FU + TD	0:39:16	935 409	43 328	736ms
FU + SSCF	1:04:39	1 490 767	91 713	4s
GC1	0:00:04	4628	78	12ms
GC1 + TD	0:01:35	102 169	10 281	1s
GC1 + SSCF	0:31:45	1 048 354	58 367	2s
AES RK + TD	0:00:07	9 439	847	61ms
AES RK + SSCF	0:09:19	410 095	6 952	195ms
AES C + TD	1:17:25	1 064 007	38 810	575ms
AES C + SSCF	1:45:00	842 583	29 770	2s

Classification one, this can be explained by the small number of attack traces containing *only* Repetitive `CCPs` (in our examples).

VI. RELATED WORK

Two main topics are related to the contributions presented in this paper: (fault injection) counter measure analysis, and cost reduction of software runtime checks.

As stated in the introduction, although numerous countermeasure schemes have been already proposed in the literature, their evaluation essentially consisted in showing how effective they are to *detect* fault injection attacks, but without precisely analyzing how to *reduce* their overhead and potential *redundancies* when applied to a whole piece of code. Moreover, most of these analysis only consider *single fault* injections. We list below some existing work on such countermeasure evaluations.

The countermeasure presented in section V-C3 is analyzed in [21] and [20]. The correctness of the hardened code and its robustness with respect to C-level *single* jump attack faults is proved by model-checking for each basic control-flow statement. In [20], a lighter version of this countermeasure scheme is proposed to reduce its overhead (at the cost of delaying the attack detection). However, this variant remains generic, and it is not (automatically) tailored for a precise code example, which could lead to further optimizations.

The countermeasure scheme mentioned in section V-C2 has been initially described in [22], and its overhead is compared to other related schemes on set of benchmarks. There is no discussion about automated way to reduce this overhead, beyond suggesting that "only critical parts of the code" need to be secured.

[23] presents an LLVM based tool allowing to evaluate software resiliency against hardware faults. It relies on the so-called SWiFi[7] technique, and relies on Klee as a symbolic test execution platform. The fault model considered are *bit-flip* of the target register of an LLVM computation or load instruction, assuming at most *one single fault*. A countermeasure mechanism is proposed by means of *executable assertions* on program variables at different code locations to detect "silent data corruptions". Experiments are provided on benchmarks, showing the results obtained in terms of fault detected, but the question of reducing the number of assertions is not addressed.

[24] focuses on software encoding schemes to protect *cryptographic implementations* against physical attacks. It proposes a metric to evaluate the robustness of an encoding scheme against *single* bit flip or instruction skip, together with a *probabilistic* simulation based evaluation method allowing to provide some robustness *vs.* performance trade-offs. Hence, it allows the user to select the most appropriate encoding scheme according to the simulation results.

Finally, [25] gives an abstract set-theoretic model to evaluate the security level brought by a mitigation scheme with respect to a given set of *exploits*. However, this model relies on a very high-level quantification on "how much" a mitigation hardens the probability that an exploit occurs (in terms of big $\mathcal{O}$ notation). Although an application example is given in the context of CFI, the granularity of the results obtained are far from the ones we can achieve in our approach.

Another line of work related to the one presented in this paper is related to the cost reduction of runtime checks in programs. Such checks are added either by the compiler to enforce type safety, or by dedicated tools like Valgrind [26] or AdSan [27], or even by assertions explicitly added by the developer. The execution time overhead induced by these extra lines of code can become quite significant and eliminating useless checks can provide important gains. Most of the existing work in this direction are based on abstract interpretation techniques (like [28]) to statically prove the validity of some assertions allowing to remove the runtime check. However, the techniques used heavily rely on the fact that the program semantic remains unchanged at runtime, which is clearly not the case in our context of fault injection.

VII. CONCLUSION

We proposed a methodology allowing to optimize a protected program with respect to a fault model and a security property within the context of multi-faults injection. Thanks to the `CCPs` side-effect free hypothesis, a trace can contain multiple `CCPs` triggered. Moreover, Lazart generates a high-order mutant embedding all possible fault injections. Hence in one single pass, we are able to deal with all combinations of faults and `CCPs`.

Associated with Dynamic-Symbolic Execution, it gives a generic approach, parameterized by the fault model and the security property, to compute the optimal combination of `CCPs` for a given `CCP`-based countermeasure.

[7]Software Implemented Fault Injection

Our experiments already show that our approach is effective on non-trivial examples, for various protection schemes. Regarding its application to *real* software products, two points need to be considered. First, our current implementation operates at the LLVM-level, which is the intermediate code representation supported by the Klee symbolic execution engine. However, from a conceptual point of view, the same operations could be performed directly on assembly code, extending dedicated DSE engines, for instance like Binsec [29] or Angr [30]. This is one direction of the future work direction we plan to investigate. Second, according to our experiments, the time complexity of our approach is dominated by the dynamic-symbolic execution step, to compute the set of symbolic attack traces and associated trigger alarms. The cost of this computation strongly depends on the considered fault model (number of of injected faults, number of injection points). The selection algorithm itself remains rather cheap, in particular when the number of execution traces with *repetitive* `CCPs` is small (as in our experiments).

Finally, another interesting future work would be to evaluate our method when considering other fault models than test inversion, like *data injection* or *function call skip* or even combination of several different fault models.

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).

REFERENCES

[1] 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.

[2] 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.

[3] 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, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds. USENIX Association, 2017, pp. 1057–1074.

[4] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A. Sadeghi, "V0ltpwn: Attacking x86 processor integrity from software," *CoRR*, vol. abs/1912.04870, 2019.

[5] 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, London, UK, November 11-15*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 195–209.

[6] 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.

[7] T. CCRA Management Committee, "Common Criteria for Information Technology Security Evaluation," Sep. 2012.

[8] "Application of Attack Potential to Smartcards and Similar Devices," Joint Interpretation Library, Tech. Rep. Version 3.0, April 2019.

[9] 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.

[10] 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.

[11] M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant, "Idea: Embedded fault injection simulator on smartcard," in *ESSoS*, ser. LNCS, vol. 8364. Springer, 2014, pp. 222–229.

[12] 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.

[13] 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.

[14] 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, Trondheim, Norway, September 21-23, 2016, Proceedings*, 2016, pp. 3–11.

[15] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[16] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *International conference on software engineering and formal methods*. Springer, 2012, pp. 233–247.

[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, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224.

[18] 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.

[19] F. de Ferrière, "A compiler approach to cyber-security," 2019 European LLVM developers' meeting, 2019.

[20] 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.

[21] J. Lalande, K. Heydemann, and P. Berthomé, "Software countermeasures for control flow integrity of smart card C codes," in *Pr. of the 19th European Symposium on Research in Computer Security, ESORICS 2014*, 2014, pp. 200–218.

[22] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[23] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Resilience evaluation via symbolic fault injection on intermediate code," in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, 2018, pp. 845–850.

[24] J. Breier, X. Hou, and Y. Liu, "Fault resilient encoding schemes in software: How far can we go?" *IACR Cryptology ePrint Archive*, vol. 2018, p. 218, 2018.

[25] R. Branco, K. Hu, H. Kawakami, and K. Sun, "A mathematical modeling of exploitations and mitigation techniques using set theory," in *2018 IEEE Security and Privacy Workshops, SP Workshops, San Francisco, CA, USA, May 24*, 2018, pp. 323–328.

[26] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation." in *PLDI*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 89–100. [Online]. Available: http://dblp.uni-trier.de/db/conf/pldi/pldi2007.html#NethercoteS07

[27] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *USENIX ATC*, 2012.

[28] N. Stulova, J. F. Morales, and M. V. Hermenegildo, "Some trade-offs in reducing the overhead of assertion run-time checks via static analysis," *Sci. Comput. Program.*, vol. 155, pp. 3–26, 2018.

[29] R. David, S. Bardin, J. Feist, J.-Y. Marion, L. Mounier, M.-L. Potet, and T. D. Ta, "Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis," in *Proceedings of SANER 2016*, Osaka, Japan, march 2016.

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.