# Combining High-Level and Low-Level Approaches to Evaluate Software Implementations Robustness Against Multiple Fault Injection Attacks

Lionel Rivière, Marie-Laure Potet, Thanh-Ha Le, Julien Bringer, Hervé Chabanne, Maxime Puys

# Combining High-Level and Low-Level Approaches to Evaluate Software Implementations Robustness Against Multiple Fault Injection Attacks

Lionel Rivière[1,2], Marie-Laure Potet[3], Thanh-Ha Le[1], Julien Bringer[1], Hervé Chabanne[1,2], and Maxime Puys[1]

[1] SAFRAN Morpho, France - first.name@morpho.com
[2] Télécom Paristech, France - first.name@telecom-paristech.fr
[3] VERIMAG, France - first.name@imag.fr
Identity & Security Alliance (The Morpho and Télécom ParisTech Research Center)

**Abstract.** Physical fault injections break security functionalities of algorithms by targeting their implementations. Software techniques strengthen such implementations to enhance their robustness against fault attacks. Exhaustively testing physical fault injections is time consuming and requires complex platforms. Simulation solutions are developed for this specific purpose. We chose two independent tools presented in 2014, the *Laser Attack Robustness* (Lazart) and the *Embedded Fault Simulator* (EFS) in order to evaluate software implementations against multiple fault injection attacks. Lazart and the EFS share the common goal that consists in detecting vulnerabilities in the code. However, they operate with different techniques, fault models and abstraction levels. This paper aims at exhibiting specific advantages of both approaches and proposes a combining scheme that emphasizes their complementary nature.

**Keywords:** Fault injection, fault simulation, instruction skipping, control flow graph, multiple fault, smartcard, embedded systems, security

## 1 Introduction

Active physical attacks, in particular fault injections, are performed against smartcard implementations in order to reveal sensitive information or break secured codes. Introduced in 1997 [1], they consist in inducing volatile faults in an operating circuit in order to generate computational errors. Several means exist to perform physical fault injections such as clock of voltage glitch [2]. Electromagnetic waves [3] and laser beams [4] improved the injection accuracy. Physical injections effects can be exploited towards Differential Fault Analysis attacks (DFA) [1,5], which aim at retrieving crucial information such as cryptographic keys by comparing faulty outputs with the correct ones. Such attacks also apply to non-cryptographic security features such as integrity checks or authentications.

In the following subsections, we describe how fault attacks threaten smartcard implementations and we propose a coarse-grained process for secure development accordingly. We emphasize actual challenges in this area and we present our contributions.

This work was partially funded by the French ANR project E-MATA HARI.

### 1.1 Fault Injection Attacks Threats

In the context of smartcard-based products, manufacturers have the challenge to ensure fault robustness for every embedded functionality. Sensitive data such as private keys are critical and must be securely managed. In order to provide confidentiality, integrity and authenticity to sensitive data, smartcard manufacturers design secure implementations, embedding software countermeasures, that are tested and evaluated against physical attacks.

However, performing an exhaustive physical fault injection evaluation would be time consuming, thus costly and therefore would come far too late in the development process. Moreover, if a vulnerability is found in the final sample code, it then remains mainly two options to smartcard developers. If a software countermeasure can handle the vulnerability, the product may be patched on already existing smartcards. However, if the vulnerability is not addressable with a software countermeasure, it could condemn the project. Furthermore, different products are built on different components. For each new product specifications, new dedicated evaluations are developed from scratch or adapted from existing ones. Hence, smartcard manufacturer need a generic method to cope with this multiplicity constraint and stem experiments complexity. This justifies the use of a global development process taking into account the robustness of the developed applications against fault injections.

### 1.2 A Coarse-grained Process

Defining a secure development process against fault injection attacks is based on the following steps:

1. identify objects that must be protected
2. develop the functional application embedding appropriate countermeasures
3. physical testing of the robustness against fault injections

This coarse-grained process must be refined in order to detect weaknesses as soon as possible. Although physical attacks are conducted on the binary code under execution, countermeasure accuracy evaluation must take place both on the source code (for instance C code) and on the assembly code (proper to each architecture). Source code robustness evaluation offers several advantages. First, a same application can be deployed on several types of architectures and C codes can then be reused, even with some adaptations depending on the component countermeasures. Then the evaluation effort is factorized between several deployments of a same application. As illustrated below, testing a low-level code against fault injection can produced a huge amount of attacks that must be examined in detail. Attacks established at the source level abstract several low-level attacks and can be more easily classified in terms of their impacts. Nevertheless, source code robustness evaluation does not give sufficient guarantees: physical attacks take place at the binary code, which can be very different than the source code (due to the compiling process). As a consequence, a low-level code analysis is also necessary. In this paper we present such a development process, based on two tools.

### 1.3 Actual Challenges

There exist several tools and approaches dedicated to test implementations against fault injections [6,7,8]. All these works differ from the fault model that is taken into account [9] and the level of code that is targeted (C or Java bytecode for the referenced works). Nevertheless all approaches face to the same problems:

- how to evaluate the dangerousness of traces obtained by fault injection
- how to compare a set of attacks
- how to establish a final verdict both in term of vulnerability or robustness

Starting from the set of assets to be protected, the first challenge consists in stating a verifiable oracle (in white or black box approach) allowing to classify attacks that jeopardize security and are not detected by countermeasures. White box oracles can be made more precise because they imply the internal state execution. Fault injection robustness is an intrinsically brute-force process, implying all possible deviant behaviors provoked by fault injections. Generally we obtain a large amount of attacks that must be reduced, to be reasonably treated. Actually there exist no criteria for that. Finally we can distinguish two classes of tools: dynamic tools start from a given execution trace which is progressively mutated [7,10] and static tools [6,8] that do not execute code and produce programs embedding some fault injections. For the first category of tools, attacks can be effectively founded but it is not possible to state robustness verdict. On the contrary, static approaches are complete but can generate false positives (for instance unfeasible paths).

We use here two complementary tools. The first one, Lazart [11], is a static tool acting on the source code and based on symbolic execution [12] that ensures both the feasibility of founded attacks and completeness verdict. The second one [10] is a low-level embedded simulator, based on a dynamic approach, which guarantee a fine-grained attack classification due to the fact that hardware mechanisms and countermeasures are not abstracted.

### 1.4 Our Contributions

- We propose a global process combining high-level and low-level robustness evaluation against multiple fault injections
- We formalize the relationship between source and assembly attacks
- Based on the complementarity between source code and assembly code attacks, we propose a systematic way to reduce the set of low-level attacks in order to facilitate the verdict statement.

After the identification of sensitive code that must be protected, we propose to conduct several evaluations :

1. Use the concolic tool Lazart in order to evaluate the code robustness to produce (or prove the absence of) high-level attacks
2. Use the low-level Embedded Fault Simulator (EFS) to produce low-level attacks
3. Evaluate the coverage of the high-level attacks by the low-level attacks
4. Evaluate the divergence between high and low-level evaluation
5. State a verdict

In Section 2, we describe the Lazart [11] approach used to perform high-level fault injection simulation and its concolic analysis capabilities. Section 3 keeps the same structure to provide a description of the low-level Embedded Fault Simulator that simulates fault effects on the assembly code on actual smartcards. We exhibit the main advantage of each approach and show how results can be compared. In complement of [10], we propose a way to classify found attacks, through the notion of *representative* attack. Section 4 illustrates the proposed process on more significant examples. In Section 4, we expose our fault simulation results on a PIN verification implementation, with the Lazart and EFS tools. Finally, in section 5, we explore the potential of combining the two tools to enhance the vulnerability detection rate and accuracy. Section 6 concludes the paper and gives some perspectives.

**A Small Example.** In the two following sections, tools are illustrated with the help of the small example given below.

```
1  // Byte array comparison
2  static bool byteArrayCompare(byte* a1, byte* a2){
3    int i       = 0;
4    int status  = BOOL_FALSE;
5    int diff    = BOOL_FALSE;
6    int size    = sizeof(a1)/sizeof(byte);
7    for (i=0; i<size; i++)
8        if (a1[i] != a2[i])
9            diff = BOOL_TRUE;
10   if ((i==size) && (diff == BOOL_FALSE))
11       status = BOOL_TRUE;
12   return status;
13 }
```

**Listing 1.1.** byteArrayCompare

Listing 1.1 describes a `byteArrayCompare` function, that compares two arrays of byte and return true if they match, false otherwise. The code of this function contains some counter-measures: for instance the test `i==size` allows to verify that we loop `size` times. '$a1$' and '$a2$' have the same size, which is checked prior to the function call. `BOOL_TRUE` and `BOOL_FALSE` are constant bytes define to `0xAA` and `0x55` respectively.

## 2 High-level Robustness Evaluation

The Lazart approach aims at evaluating the code robustness against multiple and volatile fault injections [11] and acts at the source code level (here the LLVM.3.3 intermediate representation issued from C code). The considered fault model combines an attack objective and fault injections impacting the control flow by test inversion. A test inversion consists in changing the result of a conditional jump. The originality of Lazart is to be based on a static code analysis technique, here symbolic execution. Attack paths are determined with respect to an attack objective, mastering in that the combinatorial of multiple injections. A path that fulfills the attack objective reveals a vulnerability in the code and constitutes an attack.

The main particularity of Lazart, contrary to other tools starting from a concrete trace [6,7,8], is to be able to positively or negatively qualify the result of the robustness evaluation. Thanks to the symbolic approach, Lazart explores all possible paths (and fault injection possibility) with respect to a set of symbolic inputs. Then it is possible to state if a targeted application resists to an attack of multiplicity $n$ or not.

### 2.1 The Lazart approach

The Lazart approach is based on the following steps, as illustrated on Fig. 1.

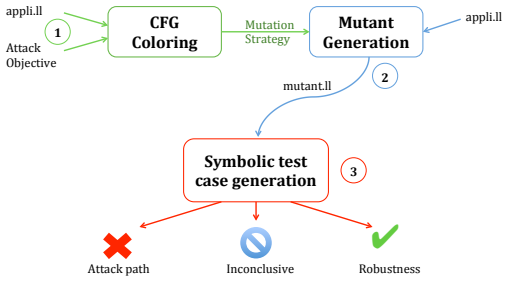


**Fig. 1.** The Lazart approach

1. Starting from the Control Flow Graph (CFG) and an attack objective, Lazart uses a reachability propagation algorithm that colors the CFG. An attack objective is a parameter that must be set, corresponding to a basic block to reach or not to reach.
2. Based on coloration, Lazart determines which program locations are candidate for successful fault injections. It produces a mutant that embeds these fault injection possibilities.

3. Using Klee, a symbolic test case generator aiming a path-coverage criterion, Lazart evaluates the robustness of the targeted application. It produces either some attacks, or establishes the absence of attacks or even an inconclusive response.

A more complete description of Lazart, and underlying algorithms, can be found in [11]. Here, we illustrate this approach on the `byteArrayCompare` example given Listing 1.1. Figure 2(a) gives the CFG associated to the `byteArrayCompare` function. The mapping between the C code and the CFG is direct, except for the test on line 10, which is split into two conditions: the basic block `for.end` testing the condition `(i==size)` followed by the basic block `land.lhs.true` testing the condition `(diff == BOOL_FALSE)`. Figure 2(b) gives the colored CFG, with the objective to reach the block `if.then9` (corresponding to the assignment `status=BOOL_TRUE` on the line 11).

A node $n$ is green whenever the attack objective is fatally reachable from $n$ and red when the objective is never reachable. The yellow color corresponds to a node from which the goal could (or not) be reachable. A yellow node owning at least one red son becomes orange. Faults are possibly injected in yellow and orange nodes, forcing to reach a green block (and consequently not fall into a red one).

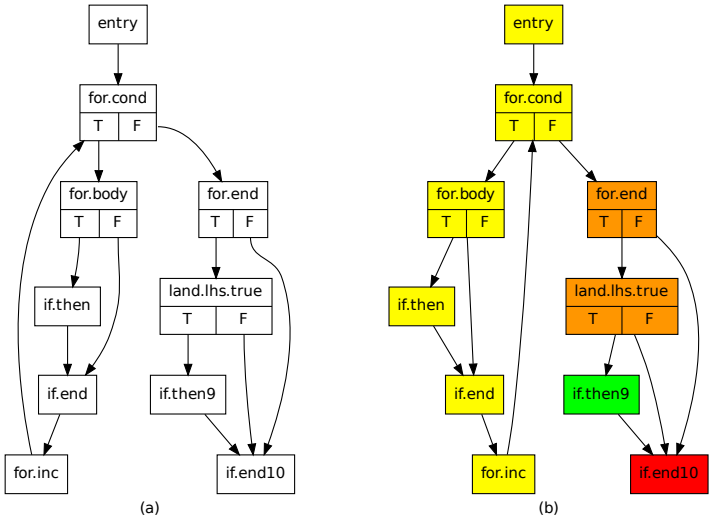


**Fig. 2.** Initial CFG of `byteArrayCompare` and after coloration

Fault injections are encoded by adding code simulating test inversions (a mutant of the initial code). A global fault number counter is added and incremented as soon as a fault is injected and each mutation is guarded by a boolean variable that indicates if the fault is injected or not. In our example, two mutations are introduced to avoid the red node `if.then10` and two double mutations hijacking the flow of the block `if.then` in the `for` loop body and another one forcing (or avoiding) the exit of the `for` loop.

Klee [13,14] is the concolic test case generation engine used by Lazart to explore all paths and thus all combinations of fault injections. It requires to declare which variables are made symbolic. Klee makes it possible to define assertions to constrain the chosen symbolic variables. The `byteArrayCompare` function takes two arrays of byte `a1` and `a2` as input arguments. Here, the size

`a1` and `a2` is set to 4, `a1` is instantiated by an initial value and `a2` is declared as symbolic with the following constraint: each byte must be different from `a1`. Variables guarding fault injections are implicitly declared as symbolic, as described below.

### 2.2 Results analysis

| Attack multiplicity (# of fault) | # of attacks | Non-redundant attacks |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 5 | 1 |
| 3 | 10 | 0 |
| 4 | 11 | 1 |
| Total | 27 | 3 |

**Table 1.** Possible attacks for byteArrayCompare

Table 1 gives the results supplied by Lazart when at most 4-faults injections are performed. An n-attack is found when the corresponding path led to reach the green block introducing $n$ faults. Attacks can be partially compared with respect to the program locations where faults are injected. An attack strictly including all locations associated to another one is considered as redundant.

For the `byteArrayCompare` function, Lazart generates 17 possibilities of fault injections. Executing the mutant that embeds these fault injections, Klee produces 56 tests in about 3 seconds. Among them, the single attack of multiplicity 1 is obtained when the loop operates normally and the fault injection forces the inversion of the condition `diff==BOOL_FALSE`. The non-redundant attack of multiplicity 2 is obtained when first, the loop is skipped (inverting the test `i<size`) and secondly, we force the condition `i==size` to be true. Others 2-faults injection attacks are redundant with the attack of multiplicity 1 (we invert one of the internal test of the body and the final condition `diff==BOOL_FALSE`). 3-faults injections do not introduce new attacks. The attack of order 4 corresponds to the case where the internal test of comparison is inverted for each byte.

The fault model considered by Lazart encompasses several data or control flow low-level attacks, depending on the compilation process: replacing an assembly instruction by no operation (`NOP`) to delete a jump or a carry flag assignment, modifying values impacting the condition, etc. Nevertheless a complete and exhaustive approach does really make sense only at the binary level where all impacts of code modification can be taken into account (such as code operation mutation). On the contrary, a coarse-grain analysis, guided by some objectives in term of sensible parts of code, takes sense during the development process where threats must be early determined and proved to be taken into account. Here we focus on control flow modification impact that is generally hard to follow in a manual audit process. Some other fault models, such as data modification, can be also simulated by code mutation as described in [15] and could be integrated into Lazart, without difficulty. Furthermore, multiple fault injection must now be taken into account according to the next state-of-the art in term of laser attacks.

Finally, the main originality of Lazart is the possibility of stating a complete verdict either in term of found attacks or absence of attacks (in our example when 0 injection is targeted). A static analysis is also used in [15,8], based on a theorem proving approach, targeting single data fault injection. The use of a concolic engine allows us either to establish the robustness or to produce attacks.

## 3 Low-Level Robustness Evaluation

The EFS approach [10] differs from Lazart [11] as it operates in runtime mode (dynamically) in the actual smartcard. No bias is introduced by external peripherals and no code mutation is performed. It uses the whole function execution time frame to exhaustively inject cycle-wise faults. For instance, without any knowledge of the code, the EFS skips every single byte of instruction, one-by-one up to $n$-by-$n$ in order to perform instruction skip even with unaligned instruction sets. The fault width $n$ is chosen and can be wider than the size of a single instruction. It is then possible to evaluate security consequences of skipping several instructions.

### 3.1 The EFS Approach

Figure 3 describes the EFS workflow. The EFS operates on the smartcard Central Processing Unit (CPU) as it is embedded in the software project with other applications. On the computer side, the EFS Handler, which consists of a software program, provides functionalities to manage fault injection experiments. The host computer is in charge of the smartcard communication.

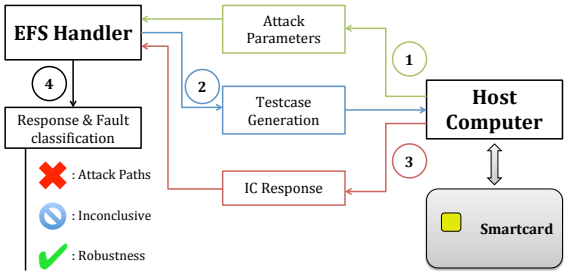


**Fig. 3.** The Embedded Fault Simulator Workflow

The whole process takes place in four main steps. First, the developer provides some attack parameters such as the fault characteristics, in term of fault model and fault width. He also specifies the targeted functionality such as PIN verification or an Rivest, Shamir, Adleman (RSA) signature. According to these parameters, in a second step, the EFS Handler calls its test case generation algorithm and produces a task list. A task consists in setting up the EFS and launching the targeted function.

At the third step, the host computer performs all tasks and sends back the functional smartcard responses to the EFS Handler. Finally, at the fourth step, the EFS handler sorts every functional responses into groups according to their functional behaviors. A functional response can be of four kinds: normal, faulty, countermeasure or CPU signal. A Functional response is said faulty under two conditions: it differs from the expected reference response and the program terminates.

In the faulty group, an attack is found when the corresponding attack path breaks the functional specification of the targeted program. For instance, a PIN verification program ensures that a PIN candidate matches the expected PIN value. Its output is binary (true of false), thus any output that invert the expected response is considered as an attack. A more complete description of the Embedded Fault Simulator can be found in [10].

### 3.2 Results Analysis

We illustrate this approach on the `byteArrayCompare` example given in Listing 1.1. As the EFS is a low-level approach, we perform our code analysis on the assembly generated from the code compilation. Chosen parameters allow to generate all paths and thus all possibilities of fault injection with respect to the instruction skip fault model.

| # of skipped instructions | # of attacks | Representative attacks |
|---|---|---|
| 1 | 4 | 4 |
| 2 | 3 | 1 |
| 3 | 2 | 1 |
| 4+ | 407 | 0 |
| Total | 416 | 6 |

**Table 2.** EFS Attacks on `byteArrayCompare`

Table 2 gives the results of the EFS according to the number of skipped contiguous instructions. In this experiment we skipped up to 16 instructions. However, 4 to 16-contiguous instructions skips, denoted as {4+} in the table, didn't reveal any new representative attacks. A detected attack is representative if it produces an unobserved functional effect leading to a successful attack.

A functional effect of an attack breaks the functional property of an instruction such as a value assignment to a register or the equality of two registers. The attack is successful if the broken assembly instruction disrupts the property of the high-level function that contains it (a comparison or a PIN Verification for instance). Several faults at the assembly level can produce the same functional effect on the high-level. We discuss this point in Section 4.4.

Three of the four attacks that skipped only one instruction are obtained when some assignment instructions are skipped (`MOV`). First, two attacks occurred at the initialization phase where `status` and `diff` should have been initialized to `BOOL_FALSE`.

```
uint8_t status  = BOOL_FALSE;
->x97A055      MOV      R10,#0x55
uint8_t diff    = BOOL_FALSE;
->x977055      MOV      R7,#0x55
```

**Listing 1.2.** Single instruction skip 1 & 2

As these two assignments are contiguous, a double instructions skip attack also succeeds but is redundant as shown in Listing 1.2. Those attacks work when `R7` or `R10` contains the precise value.

The attack effect found accounts to the same 1-fault with Lazart, which consists in inverting the condition `diff==BOOL_FALSE`.

```
if (a1[i] != a2[i])
    diff = BOOL_TRUE;
  xAD4B        CMP      R4,R11
  xD903        JE       0x38A1
->x9770AA      MOV      R7,#0xAA
```

**Listing 1.3.** Single instruction skip 3

The third single instruction skip attack avoids the update of the `diff` value when a difference is found during the comparison. This effect is showed in Listing 1.3. Consequently, the difference is not reported.

Replayed 4 times in a row, this attacks corresponds to the 4-fault attacks found by Lazart that consists in inverting the internal test of comparison for each byte.

Finally, the last single instruction skip attack is found in the sensitive part corresponding to the equality status assignment to true. Listing 1.4 gives the corresponding assembly code. At line 3, after the `CMP` (Compare), the `Z` flag (Zero) is set. In normal condition, the `JNE` (Jump Not Equal) instruction will read and reset the `Z` flag then continue without branching.

```
1 if ((i==array_size) && (diff==BOOL_FALSE))
2     status = BOOL_TRUE;
3   xAD56     CMP  R5,R6           ;(i==array_size)
4 ->xD907     JNE  0x38B2          ;return status
5   xD91F5503 CJNE R7,#0x55,0x38B2 ;diff==BOOL_FALSE
6   x97A0AA   MOV  R10,#0xAA       ;status=BOOL_TRUE
```

**Listing 1.4.** byteArrayCompare Attack

However, if we skip the line 4, the `Z` flag is not restored and will be read by the following conditional branch `CJNE` (Conditional Jump Not Equal). Consequently, even if a mismatch occurs during the array comparison, the `status` is set to `BOOL_TRUE`.

The attack effect is equivalent to the 2-fault attacks found by Lazart that consists in inverting the line 10 of Listing 1.1.

A non-redundant two instructions skip attack was found. It breaks the comparison loop by avoiding the loop counter initialization (first instruction skipped) and the loop branching routine (second instruction skipped). Consequently, the `diff` value will never be updates and will keep its `BOOL_FALSE` initial value. A non-redundant three instructions skip attack was also found but is similar to the one showed in Listing 1.4. It breaks the double condition test on the line 1 by directly branching to the `status=BOOL_TRUE` assignment after the comparison loop (line 6). This is a second manner to produce the same effect obtained by Lazart, inverting the line 10 of Listing 1.1.

Skipping 4 up to 16 instructions does not introduce new attacks. The whole process has reported 416 attacks over 3199 tests. The subset of 9 attacks consisting of $\{1, 2, 3\}$-instructions skips provided 6 representative low-level attacks. These recover the 4 non-redundant high-level attacks found by Lazart. The equivalence between the two fault attack classes is not trivial due to the abstraction level difference. The EFS performs exhaustive testing according to a fault model and with respect to the function code, size and duration. Therefore, the output fault classification allows determining the robustness of a targeted embedded application code against a chosen fault model. Moreover, output states that differ from the reference state can be evaluated to claim if the detected attack path reveals a critical vulnerability or not.

## 4 Case Study

A PIN verification algorithm constitutes a valuable target of evaluation for Lazart and the EFS tools. As it provides a secret PIN to protect, and uses a try counter to avoid brute force attacks, PIN verification is sensitive to fault attacks impacting both data and control flow. With fault injections, the aim is to break the PIN verification and/or the PIN Try Counter (PTC). In this paper, we focus on the PIN comparison. Subsection 4.1 first describes the attack scenario selected to evaluate Lazart and the EFS. Subsection 4.2 focuses on the results obtained with Lazart while part 4.3 exhibits attacks found by the EFS. A short synthesis is proposed in 4.4 to summarize both approach capabilities.

### 4.1 Secured `VerifyPIN` implementation

We analyze, in the next subsections, with both Lazart and the EFS, a secured implementation of the `VerifyPIN` functionality. First, we added the *always-decrement-first* rule [16,17,18], which recommends to decrement the PTC before any other operation occurs. When every conditional tests on the PIN passes, the PTC is incremented back. This prevents from tearing attacks, which consist in tearing the smartcard from the reader, or disabling the power just after the PIN comparison. Thereby, an attacker could not lead brute force attacks on the PIN value.

Our implementations are also prone to fault injection on data flow and control flow. The main methods to prevent fault attacks are redundancy and integrity checks. We provide the PTC integrity via backups for checking. We also managed operations involving the PTC, hence, increment and decrement are protected by checking the expected value after each operation. We also use double checks for sensitive conditional tests such as the PIN comparison itself to avoid single fault injection leading to instruction skip or test inverting [19]. The core of the `VerifyPIN` implementation is shown in Listing 1.5.

```
1  equal = BOOL_TRUE;
2  for(i=0 ; i<SIZE_OF_PIN; i++) {         // Main Comparison
3      equal = equal & ((buffer[i] != pin[i]) ? BOOL_FALSE : BOOL_TRUE);
4      stepCounter++;
5  }
6
7  if(equal == BOOL_TRUE) {
8       if(equal != BOOL_TRUE)            // Double test
9           goto counter_measure;         // Resets the remaining tries to max
10      triesLeft = MAX_TRIES;            // First backup
11      triesLeftBackup = -MAX_TRIES;     // Second backup
12      if(triesLeft != -triesLeftBackup) // Verifies the new value
13          goto counter_measure;
14      authenticated = 1;                // Authentication status update
15      if(stepCounter == INITIAL_VALUE + 4)
16          return EXIT_SUCCESS;
17 }
18 else {
19      authenticated = 0;
20      if (stepCounter == INITIAL_VALUE + 4)
21          goto failure;
22 }
```

**Listing 1.5.** `VerifyPIN` C code

| | |
|---|---|
| # of lines in C code | 83 |
| # o lines in ASM code | 67 |
| ASM Code size (in Byte) | 179 |
| Constant time comparison | √ |
| Double comparison | √ |
| Branch balancing | × |

**Table 3.** `VerifyPIN` Implementation Properties

Table 3 summarizes implementation properties. As described in section 3, we perform an exhaustive testing campaign, with respect to the targeted function size and duration. Using the EFS, the test consists in skipping every single instruction and every possible block of instructions of the very same implementation on a 16-bit smartcard. The results are described in the three following subsections.

### 4.2 Vulnerabilities Detected by Lazart

Here, we present the results produced with Lazart for the `VerifyPIN` code, where we target the block containing the statement `authenticated=1` (Listing 1.5 line 14).

```
BYTE triesLeft = maxTries;
int i;
klee_make_symbolic(buffer,
    sizeof(char)*SIZE_OF_PIN,
    "buffer");
for (i = 0; i < SIZE_OF_PIN; ++i)
   klee_assume(buffer[i] != pin[i]);
```

**Listing 1.6.** Klee Symbolic Input

The experiment aims at establishing the robustness of PIN implementations with a permissive number of trials and when the attacker does not known any part of the PIN. Then inputs are characterized for Klee as described in Listing 1.6.

`VerifyPIN` **Robustness Evaluation.** Lazart generates 6 possibilities of fault injection. Klee takes about three second to terminate normally producing 49 tests, distributed as described in Table 4, with up to 4 fault injections.

| Fault Multiplicity | # of test | # of attack | # non-redundant |
|---|---|---|---|
| 0 | 4 | 0 | 0 |
| 1 | 7 | 0 | 0 |
| 2 | 9 | 2 | 2 |
| 3 | 13 | 5 | 0 |
| 4 | 16 | 11 | 1 |
| Total | 49 | 18 | 3 |

**Table 4.** Lazart results for `VerifyPIN`

One non-redundant attack of multiplicity 2 consists in inverting the double tests `equal==BOOL_TRUE` and `equal!=BOOL_TRUE`. The other one corresponds to a fault injection that circumvents the loop execution followed by another one that hijacks the step counter value countermeasure. The attack of multiplicity 4 corresponds to the case where the internal test of the comparison is inverted for each byte.

### 4.3 Vulnerabilities Detected by The EFS

With EFS attacks on `VerifyPIN` we encountered five types of functional outputs that are described in Table 5.

| Functional Output | `VerifyPIN` |
|---|---|
| Wrong PIN / Signal or Countermeasure | 79,87% |
| Random output / APDU Errors | 18,5% |
| **Right PIN → Authentication** | **1,59%** |
| Number of tests | 4528 |

**Table 5.** Functional Output Behaviors Distribution with the EFS tool

*Wrong PIN / Signal or Countermeasure* is the most recurrent case over all experiments. *Signal* stands for illegal opcode or illegal operand and is triggered by the CPU. Software *Countermeasures* are triggered by the targeted code itself, when a fault is detected. When no CPU signal nor software countermeasures are triggered, the fault injection has no effect and leads to the *Wrong PIN* status.

Internal countermeasures are triggered when function calls or register operation are targeted (`ECALL`, `MOV`). Tampering with the `DEC/CMP` or `INC/CMP` pair also leads to trigger countermeasures. When the fault width is too large, there is a possibility to jump outside of the PIN verification execution window leading to countermeasure triggering, unresponsive smartcard, random outputs and Application Protocol Data Unit (APDU) error cases.

**`VerifyPIN` Robustness Evaluation.**

| # skipped instructions | `VerifyPIN` |
|---|---|
| 1 | 1 (1) |
| 2 | 2 (1) |
| 3 | 1 (0) |
| 4 | 4 (0) |
| 5+ | 64 (1) |
| Total | 72 (3) |

**Table 6.** Detected Attacks by the EFS on `VerifyPIN`

Table 6 describes the fault obtained attacking `VerifyPIN`, according the number of skipped instructions. From the 72 successful attacks over 4528 tests (1,59%), there are only 2 representative vulnerabilities found in the code under 4-instructions skips. The varying fault width explains this rate. The fault width corresponds to the number of skipped bytes of opcode. Most successes are explained by skips of conditional tests implying the `triesLeft` counter or the `BOOL_TRUE` value.

If we consider only addresses where successful attacks started from, we obtain the following table 7. It describes the fault width according to the faulted assembly code address that led to successful

authentications. The digit between parentheses corresponds to representative attacks. Several non-representative $\{1, 2\}$-instructions skip faults were found. Those results are exposed with more details on the following subsections.

| Address (ASM code) | Fault width | #instr. | Corresponding C code |
|---|---|---|---|
| 0x4909 | 0x16 | 9 | `If (triesLeft ≤ 0)` |
| 0x490D | 0x12 | 7 | |
| 0x490F | 0x10 | 4 | `If (t1 != triesLeftBackup)` |
| 0x4914 | 0x0B | 6 | |
| 0x4970 | 0x08 | 4 | `If (equal == BOOL_TRUE)` |
| 0x4974 | 0x04 | 1 | |

**Table 7.** Faulted Code Address in `VerifyPIN`

In the #instr. column we show the corresponding number of skipped instruction. Successful attacks imply to skip up to 9 contiguous instructions, which is hard to achieve with a physical injection.

Moreover, Table 7 exhibits one attack path that only requires a single instruction skip. The corresponding C code is the conditional test on the boolean value `equal` that states the equality of the two compared PINs (lines 1-5 in Listing 1.1, recalled in Listing 1.7 below).

```
equal   =   BOOL_TRUE;
for (i=0; i<SIZE_OF_PIN; i++) {             // Main Comparison
    equal = equal & (buffer[i] != pin[i]) ? BOOL_FALSE : BOOL_TRUE);
    stepCounter++;
}
```

**Listing 1.7.** `VerifyPIN` C code

The `equal` value is used as a part of the comparison at each step, for each byte comparison.

A single difference between the PIN value and the input PIN is sufficient to switch the `equal` value to `false`. A single instruction skip leading to an authentication with a wrong input PIN at `equal == BOOL_TRUE` is not easy to detect at the C level. The analysis has to be pushed further, at the assembly level. While reading the assembly code, we notice that neither the byte-to-byte comparison nor the PIN values is altered during the `for` loop. However, the `equal` value is initialized to the constant value `BOOL_TRUE` before the loop starts as stated in Listing 1.7 in the C code.

```
equal   =   BOOL_TRUE;
  x9710         MOV       R11,#DWR(0x10)
  x9741000A     MOV       EQUAL(0x000A),R11
```

**Listing 1.8.** `VerifyPIN` ASM `equal` Assignment

In Listing 1.8 we show how the `equal` assignment takes place at assembly level. First, the `BOOL_TRUE` constant is copied from its address `#DWR(0x10)` to the register `R11`.

Then, `R11` is immediately copied into `EQUAL(0x000A)`, which stands for the `equal` variable in the C code. During the loop, `EQUAL(0x000A)` takes part in a multi-conditional assignment formula. The `R11` register is still unused during the loop. At the end of the loop, `EQUAL(0x000A)` is written back to `R11` and compared to the `BOOL_TRUE` constant as shown in Listing 1.9.

```
1 ->x97D1000A    MOV     R11,#EQUAL(0x000A)
2   xD9101B      CJNE    R11,#DWR(0x10),AUTH(0x4996)
3   xD91024      CJNE    R11,#DWR(0x10),CTM(0x49A2)
```

**Listing 1.9.** `VerifyPIN` ASM code

Consequently, if the `MOV` responsible of that copy is skipped (line 1), the `R11` register will not be updated and will keep its latest value, namely `BOOL_TRUE`.

Therefore, this single skip leads to a successful authentication without disrupting the control flow.

### 4.4 Synthesis

We encountered different behaviors during our experiments. Less than 2% of the EFS attack paths against `VerifyPIN` implementations leads to a successful authentication. Most of reported attacks are redundant. They include all attacks detected by Lazart but except 4-faults injections due to the fact that, in the current implementation of the EFS, only contiguous instructions are skipped. However, high-multiplicity faults performed by Lazart can target different lines of the C code, at different locations. This is why those two faults are not reported by the EFS. However, for each other non-redundant attack found by Lazart, there is at least one attack found by the EFS that reflect the same code vulnerability. In particular, for a given vulnerable code line detected by Lazart, there can be different explanations in the underlying assembly code, which are reported by the EFS.

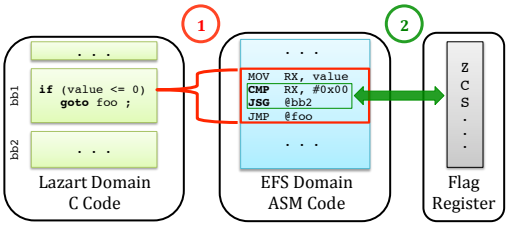


**Fig. 4.** Fault Level Differences Insights

As shown on Figure 4, the first observation is that, to a statement at high-level C code corresponds several low-level assembly code lines (1). Secondly, some low-level operations are totally abstracted at the C-level and thus, they cannot be targeted. Some instructions induce implicit operations, such as flag register read/writes. To have a good insight of flag states updates (2), the developer must consider the dynamic assembly code behavior.

Therefore, vulnerable instructions such as `CMP RX,#0x00` can be avoided. Regarding instruction skips, a careful attention must be paid to manage branching codes in order to avoid unexpected branching attacks, as illustrated in the example given in Listing 1.4.

## 5 Combining Lazart and the EFS to Improve the Vulnerability Detection

In this section, we expose some interesting results obtained by combining Lazart and the EFS tool in evaluating both the `VerifyPIN` and the `byteArrayCompare` implementations. The two approaches reveal some complementary despite their different operating mode. On the one hand, Lazart can locate a sensitive portion of code at the C level according to an attack goal but it does not consider the underlying mechanism. On the other hand, the EFS can perform exhaustive instruction skipping in a whole function time frame but without any predetermined security purpose. Performances, in terms of timing and accuracy differ according to the security property that the developer is trying to evaluate or reach.

Here, the main idea is to combine both approaches in order to perform a low-level exhaustive testing on assembly codes that correspond to some sensitive code blocks determined by Lazart. We proceed as described below:

1) Determine the attack goal with Lazart. In our case study, we must ensure that the PIN authentication fulfills its security functionality.
2) Locate sensitive portions of code according to the CFG coloring algorithm, the green basic block. If it is reached under fault injection, the goal no longer holds.
3) Setup the EFS code range target according to the assembly code address range that corresponds to the green basic block in the source code.
4) Perform the EFS within the restricted code area

As we first try Lazart and the EFS independently on `byteArrayCompare` and `VerifyPIN`, we propose to run the combined tests on those two implementations. To do so, we define two benchmark criteria. First, the *Detection rate* denotes the capacity of an approach to reveal distinct vulnerabilities in the code via the non-redundant attacks. It is the ratio between the number of attacks found and the number of non-redundant attacks. Secondly, the *Timing performance* criterion denotes the time spent to perform the whole process and is largely related to the total number of test.

In the rest of this section, all results are obtained under the following attack parameters. Lazart performed multiple fault injections up to four faults. The EFS performed up to four contiguous instructions skips for all implementations. Non-redundant and representative attacks are reported in parentheses.

| Approach | # of tests | # of attacks | Detection rate | Timing performance |
|---|---|---|---|---|
| Lazart | 56 | 27 (3) | 11,7% | $\sim$ 3s |
| EFS | 2652 | 204 (6) | 2,9% | $\sim$ 9mn |
| Lazart + EFS | 56 + 572 | 20 (4) | 20% | $\sim$ 2mn |

**Table 8.** Lazart & EFS Complementarity Results on `byteArrayCompare`

Table 8 describes the results obtained by the combination of Lazart and the EFS on the `byteArrayCompare` function. Chaining Lazart and the EFS greatly improves the detection rate and is 4,5 times faster than the EFS alone. There is a difference of magnitude between the Lazart and the EFS timing performance. This arises because of the operating platform. Lazart runs on powerful x86 processors clocked at several GHz whereas the EFS runs at best on 33MHz smartcards.

Lazart performs its fault injection simulation based on code mutation. Each mutation consists in forcing a conditional test to branch or not to branch, regardless values considered in the targeted test. Thereby, whatever the conditional test, the fault will occur. However, as described in Listing 1.2, two attacks are found in the initialization phase of the `byteArrayCompare` function. Those two are not reported because Lazart does not take data mutations into account. Consequently, at the low level, the EFS alone found two non-redundant faults that are not reported with the combined approach. This highlights the importance of tracking the evolution of values that are used in conditional tests. We simply extended the portion of code spotted by Lazart with the portion of code where sensitive values are manipulated for the combined approach. Therefore, the combined approach is able to retrieve all six attacks.

| Approach | # of tests | # of attacks | Detection rate | Timing performance |
|---|---|---|---|---|
| Lazart | 49 | 18 (3) | 16,6% | $< 3$s |
| EFS | 4528 | 72 (2) | 2,7% | $\sim$ 17mn |
| Lazart + EFS | 49 + 720 | 14 (3) | 21,4% | $\sim$ 1mn30s |

**Table 9.** Lazart & EFS Complementarity Results on `VerifyPIN`

Table 9 describes the results obtained on `VerifyPIN` code. The combination reduces the experiment duration by a factor of 10 thanks to the range reduction operated by Lazart. Moreover, the detection rate increases, it reflects a more accurate fault detection. Lazart and the EFS found two different sets of attacks in the same code area. This explains the better non-redundant attack detection rate.

Lazart and the EFS are two fault injection simulation tools that operate at different abstraction levels with different fault models. However, we proposed a new method to make them work together. Our experiments highlight the interest of combining them despite of their differences and show significant performance improvements. The high-level static approach helps to refine the code range of the low-level dynamic one, and altogether they improve the vulnerability detection rate of our fault simulation.

## 6 Conclusion

In this paper, we study the application of two simulation techniques to evaluate smartcards robustness against multiple fault attacks and show their complementary. Lazart performs concolic analysis on the control flow graph of a C code under the test inverting fault model. The EFS performs exhaustive instruction skip at the assembly level with respect to a targeted function. We exposed our attack results on a common PIN verification implementation. We show that software countermeasure implementations must be tested to avoid coding errors or some compiler optimizations that could lead to the countermeasure deprecation. We also took advantage of Lazart and the EFS differences by combining them. It results in a new multi-level fault injection simulation tool that improved the fault vulnerability detection rate and accuracy.

There exist several tools and approaches dedicated to test implementations against fault injections [6,7,8]. All these works differ from the fault model that is taken into account and the level of code that is targeted (C or Java bytecode for the referenced works). Actually, there exists no criteria allowing us to evaluate and compare such approaches. The work presented here constitutes a first step in this sense: we propose some criteria based on the number of generated tests and non-redundant attacks (number of generated attacks is not significant). Then we proposed to evaluate the efficiency of a test campaign against fault injection by a detection rate. Proposed criteria must be refined and extended for instance in taking into account coverage criteria. Another contribution of this work is the proposition of a method to combine high and low level evaluations. In [7], the authors exploit high-level attacks generation in order to evaluate a low level simulation. But it is an *a posteriori approach*. Here we propose to combine *a priori* fault injection simulations. A finer analysis must be conducted in order to estimate the gain of this combination.

## References


1. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In W. Fumy, editor, *EUROCRYPT*, volume 1233 of *LNCS*, pages 37–51. Springer, 1997.
2. J. Balasch, B. Gierlichs, and I. Verbauwhede. An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In L. Breveglieri, S. Guilley, I. Koren, D. Naccache, and J. Takahashi, editors, *FDTC*, pages 105–114. IEEE, 2011.
3. A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria. Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES. In G. Bertoni and B. Gierlichs, editors, *FDTC*, pages 7–15. IEEE, 2012.
4. S. P. Skorobogatov and R. J. Anderson. Optical Fault Induction Attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *LNCS*, pages 2–12. Springer, 2002.
5. E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In Burton S. Kaliski Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.
6. J.-B. Machemie, C. Mazin, J.-L. Lanet, and J. Cartigny. SmartCM a smart card fault injection simulator. In *WIFS*, pages 1–6. IEEE, 2011.
7. P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky, and J.-F. Lalande. High Level Model of Control Flow Attacks for Smart Card Functional Security. In *ARES*, pages 224–229. IEEE Computer Society, 2012.
8. M. Christofi, B. Chetali, L. Goubin, and D. Vigilant. Formal verification of a CRT-RSA implementation against fault attacks. *J. Cryptographic Engineering*, 3(3):157–167, 2013.
9. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, Feb 2006.
10. M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant. Idea: Embedded Fault Injection Simulator on Smartcard. In J. Jürjens, F. Piessens, and N. Bielova, editors, *ESSoS*, volume 8364 of *LNCS*, pages 222–229. Springer, 2014.
11. M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil. Lazart: a symbolic approach for evaluation the robustness of secured codes against control flow fault injection. In *ICST*, 2014.
12. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
13. The KLEE symbolic virtual machine. http://klee.llvm.org/.
14. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, pages 209–224, 2008.
15. M. Christofi. *Preuves de sécurité outillées d'implémentation cryptographiques*. PhD thesis, Laboratoire PRiSM, Université de Versailles Saint Quentin-en-Yvelines, France, 2013.
16. J. Uguchi-Cartigny A. A.-K. Sere, J.-L. Lanet. Carte à puce Java Card : Protection du code contre les attaques en faute, 2009.
17. L. Folkman. *The use of a power analysis for influencing PIN verification on cryptographic smart card*. Bakalásk práce, Masarykova univerzita, Fakulta informatiky, 2007.
18. D. Sauveron. *Etude et réalisation d'un environnement d'exprimentation et de modélisation pour la technologie Java Card : application à la sécurité*. PhD thesis, Université Bordeaux 1- Informatique et Mathématiques, 2004. Thèse de doctorat dirigée par Chaumette, S.
19. J. G. J. van Woudenberg, M. F. Witteman, and F. Menarini. Practical Optical Fault Injection on Secure Microcontrollers. In L. Breveglieri, S. Guilley, I. Koren, D. Naccache, and J. Takahashi, editors, *FDTC*, pages 91–99. IEEE, 2011.