2014 IEEE International Conference on Software Testing, Verification, and Validation

Lazart: a symbolic approach for evaluating the robustness of secured codes against control flow fault injections

Marie-Laure Potet
Verimag
University of Grenoble
Marie-Laure.Potet@imag.fr

Laurent Mounier
Verimag
University of Grenoble
Laurent.Mounier@imag.fr

Maxime Puys
Verimag
University of Grenoble
Maxime.Puys@imag.fr

Louis Dureuil
Verimag
University of Grenoble
Louis.Dureuil@imag.fr

***Abstract*—[1] In the domain of smart cards, secured devices must be protected against high level attack potential [1]. According to norms such as the Common Criteria [2], the vulnerability analysis must cover the current state-of-the-art in term of attacks. Nowadays, a very classical type of attack is fault injection, conducted by means of laser based techniques. We propose a global approach, called Lazart, to evaluate code robustness against fault injections targeting control flow modifications. The originality of Lazart is twofolds. First, we encompass the evaluation process as a whole: starting from a fault model, we produce (or establish the absence of) attacks, taking into consideration software countermeasures. Furthermore, according to the near state-of-the-art, our methodology takes into account multiple transient fault injections and their combinatory. The proposed approach is supported by an effective tool suite based on the LLVM format [3] and the KLEE symbolic test generator [4].**

***Keywords*-symbolic test generation, fault injection by mutation, smart card vulnerability analysis**

I. Introduction

A. Context

Secured devices such as smart cards, security tokens, and in a near future mobile phones, are submitted to drastic secure requirements and certification process. In the domain of smart card, secured devices must be protected against high level attack potential as described in [1] (such as multiple attackers with a high level of expertise, using sophisticated equipments, etc.). Then, according to norms such as the Common Criteria [2], the vulnerability analysis must cover the current state-of-the-art in term of attacks[2]. Nowadays, a very classical type of attack is fault injection, that can be conducted by techniques such as laser attacks [5]. These attacks consist in modifying some hardware components (memory or buses) in order to influence the current execution to force, or avoid, some sensitive operations (such as a pin verification or the generation of a new random number). In complement with classical hardware countermeasures, codes are hardened by software countermeasures (managing integrity counter, redundant conditions, etc.). Vulnerability analysis requires to take into account faults that can be injected, their logical impacts, and corresponding countermeasures embedded in the application code.

B. Fault model and robustness evaluation

A classical approach for fault injection consists in defining an appropriate fault model and in evaluating the robustness of the code relatively to this fault model.

1) Binary level fault model: According to the state-of-the-art, fault models have been proposed for laser attacks [5], [6]. The more realistic model consists in changing a value from 0x00 to 0xFF, or from 0xFF to 0x00 or by a random value (for encrypted memory). These modifications can apply either at the level of one bit, or one byte or a group of bytes. In general a laser attack impacts the code of applications stored in EEPROM or the data passing through the buses. As pointed out in [1] laser attack effects consist in modifying a value read from memory or modifying the execution flow in various ways (inverting a test, generating a new jump or a calculation error, etc.). Complementary with brute force approaches simulating laser attacks at the binary level, some works ([7], [8]) propose high level attack models (generally at the source code level), modelling the possible impacts of attacks. For instance [8] targets variable modification attacks, whereas in [7] the authors model attacks replacing an instruction by a NOP opcode or changing the destination address of a branch instruction.

2) Permanent versus volatile fault injection: Depending where the code is stored, and how laser attacks are conducted, two different consequences must be considered. *A permanent error* corresponds to an effective modification of the program code. *A volatile (or transient) fault* corresponds to a fault injection during a run [9]. Permanent errors do not appear to be a very realistic model: current smart card platforms generally contain some system countermeasures to detect the loading of modified programs (for instance with a checksum verification). In practice, attacks are conducted during code execution by introducing volatile faults. Furthermore, whereas until recently the techniques in term of laser attack reduced to a single shot per run, several spatial or temporal attacks must now be considered as plausible. Spatial multiple attacks consist in modifying two independent binary values, temporal

[1]This work has been partially supported by the LabExPERSYVAL-Lab(ANR-11-LABX-0025)

[2]We aim here the AVA class, dedicated to vulnerability assessment

978-0-7695-5185-2/14 $31.00 © 2014 IEEE
DOI 10.1109/ICST.2014.34

multiple attacks consist in modifying the same value several times during a run. Thus, a leading-edge approach in term of fault injection analysis must encompass volatile and multiple fault injections.

3) Robustness evaluation: Evaluating the robustness of an application against fault injection can be seen as the production and execution of a set of mutants, corresponding to all faulty behaviours [10]. These mutants can be produced dynamically or statically. When a dynamic approach is chosen, an execution is launched and faults are injected during a run, according to a given fault model. Dynamic mutation approach faces with the classical incompleteness problem: depending on how inputs are selected, it is difficult to quantify the robustness of the considered application. When mutants are statically produced, they have to be classified using criteria as proposed in [11]: their dangerousness (in term of elevation of privilege for instance) and the presence of countermeasures detecting the corresponding attack. In both cases, if multiple fault injections must be taken into consideration, we are faced with the problem of a combinatorial explosion.

4) Development and Certification processes: Assisting the robustness evaluation process against attacks could be helpful both during the development phase (handled by the card manufacturer) or during the certification phase (handled by well-identified third-party security labs[3]). In both cases, developers or evaluators must acquire a fine-grained understanding of code in particular to evaluate the relevance of software and hardware countermeasures. Source or assembly codes appear to be a well-adapted level to do that. Finding attacks at the source code can help vulnerability analysis in complement to a low level brute-force approach [7] consisting in dynamically mutating bits, bytes or groups of bytes, in particular when some well-identified dangerous scenarii must be examined.

C. The Lazart approach

The approach we proposed, called Lazart, targets code robustness evaluation against multiple and volatile fault injections. We start with a high level fault model, combined with the identification of sensitive statements, allowing us to produce (or establish the absence of) attacks. We address fault injections directly impacting the control flow and then modifying the logic of the applications, in particular attacks by test inversion (changing the result of a conditional jump). This fault model is a realistic one due to the fact its encompasses several data or control flow low level attacks [12] (introducing a NOP to delete a jump or the assignment of a carry flag, modifying values impacting the condition, etc.). Furthermore we do not focus on cryptographic algorithms, generally mainly sensitive to data mutation, but we target any secured applications such as authentication and identification processes, digital rights management codes or banking exchange transactions. Security weaknesses of these applications can result on control flow modifications implying for instance verifications of identification values, current state variables or the structure of inputs data.

[3]Such as Cesti in France.

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

1) Starting from identified sensitive statements (attack objective) we compute structural information on the control flow graph (cfg), corresponding to some reachability properties.
2) Using previous information we determine which program locations are candidate for fault injection and we produce a unique corresponding mutant.
3) Using a symbolic test case generator and a path-coverage criterion we evaluate the robustness of the application producing either some attacks, or establishing the absence of attacks, or an inconclusive response.

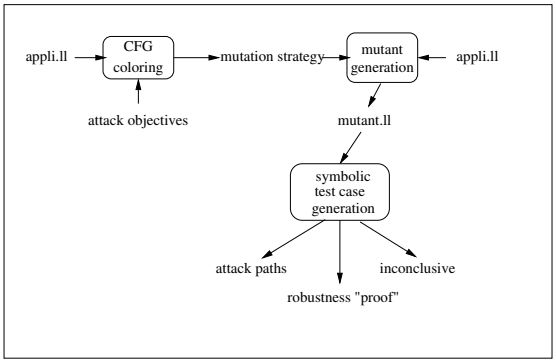

Fig. 1. The Lazart approach

The main innovative part of Lazart is to encompass the entire process from a fault model, to the production (or proof of absence) of attacks. We also take into consideration the dangerousness of attacks, i.e. fault injections producing an execution not stopped by some embedded software countermeasures and reaching a sensitive objective. The second contribution is to address multiple volatile fault injections, without producing a combinatorial number of mutants. Finally Lazart implementation works on a (well-known) intermediate representation, the LLVM format [3] and uses KLEE [4], [13], a concolic test generator engine taking as input LLVM codes (files appli.ll). Contrary to source code as C ([7], [8]), an intermediate format offers a fine enough granularity level for expressing mutants. Furthermore, it makes easier the relation between high level and binary level fault models (this later one corresponding to the real laser attack process).

Section II describes the cfg coloring algorithm computing reachability properties. Section III explains how the colored cfg is exploited to determine mutations and gives the algorithms producing LLVM mutants. Section IV explains how symbolic test generation can be used to evaluate the robustness of an application and the instanciation of this method using KLEE. Section V proposes a way to classify attacks, presents the Lazart tool suite and some experimental results. Finally, section VI compares the Lazart approach with other works and gives some perspectives on this topic.

II. Control flow graph reachability analysis

The first step of the Lazart approach is based on a reachability analysis performed on the cfg of the application. We introduce a simple example to illustrate this analysis.

A. A simple example

The example we consider consists in a pin verification algorithm presented in [14]. Starting from a naive code (sensitive to channel and laser attacks), we will see in section V a more robust implementation. Listing 1 is the C code of the naive version (function `Verify`) and Fig 2(a) presents the corresponding control flow graph produced by LLVM [3]. Basic blocks are identified in Listing 1 by means of comments and Table I gives the mapping between LLVM block names and C code lines.

```
1 #define SIZE_OF_PIN 4
2 #define maxTries 3
3 typedef unsigned char BYTE;
4 BYTE triesLeft = maxTries;
5 BYTE authenticated = 0;
6 BYTE pin[4] = {(char)1, (char)2, (char)3, (char)4};
7
8 BYTE Verify(char buffer[4]) {
9         BYTE i;
10 // BLOCK entry
11         // No comparison if PIN is blocked
12         if(triesLeft <= 0) goto FAILURE;
13 // BLOCKS  bb (initialisation), bb4 (loop condition)
14         // Main Comparison
15         for(i = 0; i < 4; i++)        // BLOCK bb3 (i++)
16 // BLOCK bb1
17                 if(buffer[i] != pin[i]) {
18 // BLOCK bb2
19                         triesLeft--;
20                         authenticated = 0;
21                         goto FAILURE;
22                 }
23 // BLOCK bb5
24         // Comparison is successful
25         triesLeft = maxTries;
26         authenticated = 1;
27         return EXIT_SUCCESS;
28 //BLOCK FAILURE
29     FAILURE : return EXIT_FAILURE;
30 //BLOCKS bb6, return: exit with  the return value
31 }
```

Listing 1. A naive implementation of Verify

entry	bb	bb1	bb2	bb3	bb4	bb5	bb6	FAILURE
10	13	16	18	15	13	23	30	28

TABLE I
Mapping between block names and C code

B. Basic blocks coloring

The inputs of the coloring algorithm are a cfg and a set of basic blocks identified as "sensitive", either because we want to trigger their execution (block "to be reached"), or to circumvent it (block "not to be reached"), from the attacker point of view. For instance, on Listing 1, we may want to trigger the execution of assignment `authenticated = 1`

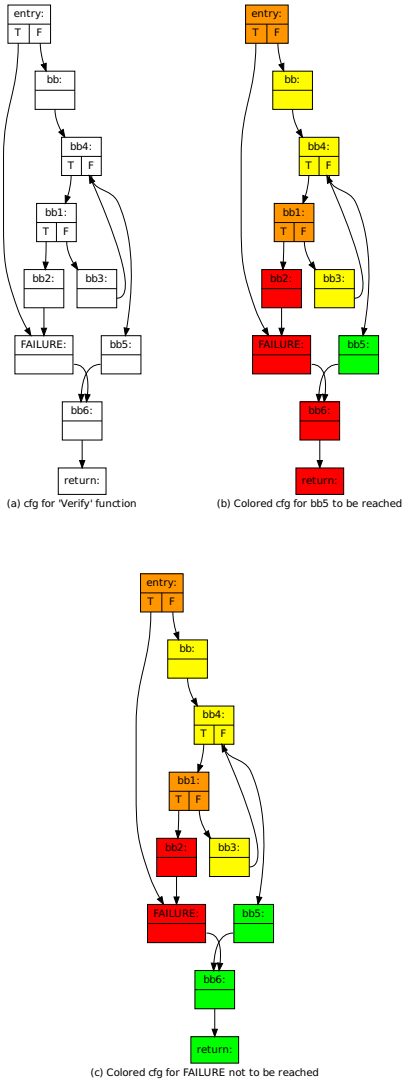

Fig. 2. Initial cfg of `Verify`, and two colored graphs

(in block bb5), or to circumvent a failure detection (i.e. not executing the FAILURE block). Generally speaking, we want to trigger the execution of instructions allowing us to gain privileges and we want to circumvent executions corresponding to countermeasures.

We denote a cfg as a graph (N, E), where N is a set of nodes[4] and E the set of edges. $Sons(n)$ is the set of direct sons of a node n with respect to E, and $Leaf$ is the set of leaf nodes. An *attack objective* O is an element G of N, with the indication "to be reached" or "not to be reached". Definitions 2.1 and 2.2 give the coloring algorithm of N depending on O.

[4]each node corresponding to a basic block

Definition 2.1: cfg coloring for O =(G, "to be reached").

- Green is the set of nodes that will eventually reach node G. It is defined as the smallest set verifying:
$$Green = \{G\} \cup \ (\{n : Sons(n) \subseteq \ Green\} \backslash Leaf)$$
- Red is the set of nodes that can not reach G. It is defined as the greatest set verifying:
$$Red = \{n : Sons(n) \subseteq Red)\} \backslash \{G\}$$
- Yellow is the set of nodes that can or cannot reach G, depending of the execution flow:
$$Yellow = N \backslash (Green \cup Red)$$

Definition 2.2: cfg coloring for O =(G, "not to be reached"). When unreachability is targeted, sets $Green$ and Red in Def. 2.1 are inverted.

Definition 2.3: Orange node. We define a subset of Yellow as yellow nodes with a red son:
$$Orange = \{n : n \in Yellow \wedge \ (Sons(n) \cap Red \neq \emptyset)\}$$

The Lazart tool suite implements the block coloring algorithm according to the previous definitions. Figure 2 illustrates the results obtained from this algorithm for two objectives: forcing the execution of bb5 (Fig. 2(b)) and circumventing the execution of FAILURE (Fig. 2(c)). Consider the following mapping if colors are not available: green=white, red=black, yellow=light grey and orange=dark grey.

III. Multiple Fault injections and mutants

Based on the colored control flow graph, we define a strategy, called Comp, introducing all possibilities of fault injections by test inversion targeting the attack objective. Then, we explain how it can be enforced through mutants.

A. A Fault injection strategy for test inversion

Table II describes how faults are injected, depending on the current block color. Intuitively, when we are executing a green block B, the attack objective is satisfied (all paths issued from B reach a node "to be reached" or circumvent a node "not to be reached"). Conversely, if B belongs to Red, the attack objective always fails (all paths issued from B do not reach the expected block or always reach the unexpected block). Thus, faults (i.e., test inversion) need to be injected only on yellow or orange nodes: we avoid red nodes (orange node case of table II), we favour green node (second to last case of table II) and otherwise we explore the two consequences of fault injection (last case of table II). As a consequence, strategy

Node (with two sons)	Action
Green	do nothing
Red	unreachable
Orange	force the test inversion to circumvent the red son
Yellow with a green son	possibly invert the test to reach the green son
Yellow with two yellow nodes	possibly invert the test from one son to the other one and vice-versa

TABLE II
Comp: A fault injection strategy

Comp always ensures that all red blocks will be circumvented. Then, if the code execution with faults injection terminates, a green block will eventually be reached[5]. We call *mandatory mutation* when we force a test inversion (on an orange node), and *optional mutation* when the test inversion could take place or not (on a yellow node).

Example 3.1: Comp Strategy effect on the function `Verify`. If we consider the control flow graph of Fig. 2(b) we obtain the following mutation points (the same result would be obtained for Fig. 2(c)):

```
File Verifybb5Comp:
   Mandatory mutations:
      entry: FAILURE/bb, bb1: bb2/bb3
   Optional mutations:
      bb4: bb1/bb5
```

Notation `bb: bb1/bb2` means that the block called `bb` must be muted to hijack the control flow from block called `bb1` to block called `bb2`.

B. Mutants and multiple fault injections

Existing approaches consisting in building mutants for laser attacks generally consider a single fault injection [15], [7]. Then, fault injection is encoded into mutants by explicitly modifying the control flow, without taking into account the flow induced by the current inputs. Fig 3 describes such a transformation (colors are lost by mutation). Our objective being to take into account *multiple* fault injections, we could not reasonably produce one mutant per each possible combination of fault injections. Thus, we produce a single mutant encoding all possible fault injections (sometimes called Higher Order Mutant [16]). To do so, we introduce a particular variable, called `fault`, which is incremented each time an attack effectively takes place. Furthermore, we use a lazy approach in which faults are not necessarily injected if the current flow follows a winning path. With this solution, we exploit at the best the interaction between possible inputs and fault injections, in order to minimize the number of laser shots.

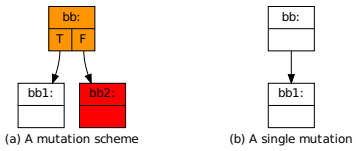

Fig. 3. Encoding Mutation independently of the current inputs

C. Mandatory mutation operator

Mandatory mutation operator enforces the hijack of the control flow just before a red node is going to be reached, as described on Fig. 4. Contrary to Fig. 3, a fault is injected only if the current flow does not follow a possibly winning path. The new block `bbTI` is introduced to count the number of fault injection (see Listing 2).

The mandatory mutation is implemented in LLVM as described on Listing 2. In LLVM, a conditional branch is of the form `br i1 %Cond, label %bbTRUE, label`

[5]Supposing the objective is connected to the entry point.

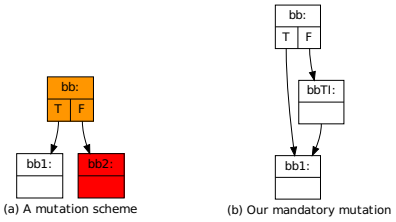

Fig. 4. Mutation operator when forcing a test inversion

`%bbFALSE`, where `%Cond` represents a 1-bit value. If this value is evaluated to true (resp. *false*), control flows to the `%bbTRUE` label argument (resp. `%bbFALSE`). The original `br` instruction of the block we want to mutate (line 4) is replaced by the one hijacking the control flow toward the `bbTI` block (line 5) (semicolon introduces LLVM comments).

```
1 bb:
2   ... ;Block to mutate
3
4   ;br i1 %Cond, label %bb1, label %bb2 ; Original condition
5   br i1 %Cond, label %bb1, label %bbTI ; Mutated condition
6
7 bbTI:
8   %mut_1 = load i32* @fault, align 1      ; Increments the
9   %mut_2 = add i32 %mut_1, 1              ; global variable
10  store i32 %mut_2, i32* @fault, align 1 ; @fault
11  br label %bb1 ;
```

Listing 2. LLVM mandatory mutation pattern

D. Optional mutation operator

In order to control the optional mutations (i.e. faults introduced on a yellow node, see Table II), we add an extra boolean variable (named `activX`) which must be valuated to 1 to inject a fault. Fig. 5 explains how this mutation operates, transforming Fig. 5(a) into Fig. 5(c). The two new blocks `T1bb1` and `T1bb2` control the introduction of faults, depending on the value of variables `activbb1` and `activbb2` and the two blocks `T2bb1` and `T2bb2` increment the fault number.

If `bb2` is a green node the transformation 5(b) applies (symmetrically if `bb1` is green and `bb2` yellow). In Listing 3 we give the LLVM code associated to blocks `T1bb1` and `T2bb1`.

```
1 T1bb1:
2   %mut_1 = load i32* @activbb2, align 1    ; test on the
3   %mut_2 = icmp eq i32 %mut_1, 1           ; value of
4   br i1 %mut_2, label %T2bb1, label %bb1   ; @activbb2
5
6 T2bb1:
7   %mut_3 = load i32* @fault, align 1     ; Increments @fault
8   %mut_4 = add i32 %mut_3, 1
9   store i32 %mut_4, i32* @fault, align 1
10  br label %bb2
```

Listing 3. Optional mutation pattern

E. Mutant generation

Starting from an LLVM application code, a file `name.ll`, and a file describing which nodes must be muted and how (see example 3.1), we produce a mutant, named `NameMutComp.ll`. Mutant generation is implemented into a tool called Wolwerine (see Section V-B).

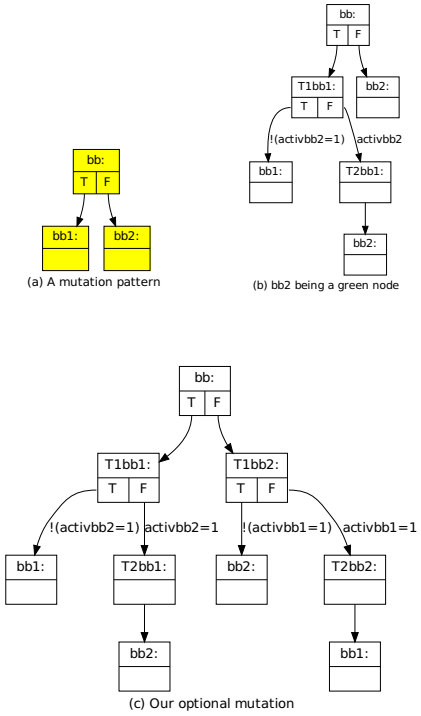

Fig. 5. Mutation operator when possibly inverting a test

The mutant `VerifyMutComp.ll`, expressed as a C code to be more readable, is given on Listing 4. According to example 3.1, two mandatory mutations are introduced to avoid the two red blocks `bb2` and `FAILURE` (code corresponding to these two blocks has been deleted). The local variable `activbb5` guards the activation of the fault consisting in forcing the loop exit (optional mutation `bb4: b1/bb5`). Function `klee_make_symbolic` (line 8) can be seen here as a function assigning any value to the variable `activbb5` (see Section IV-A for more explanations).

```
1 BYTE Verify(BYTE buffer[4]) {
2   int i=0; fault=0;
3 // Mandatory mutation on entry: FAILURE/bb
4   if (triesLeft <= 0) fault++;
5 // Optional mutation on bb4: bb1/bb5
6   while (i < 4) {
7     int activbb5 ;
8     klee_make_symbolic(&activbb5,sizeof(int),"activbb5");
9     if(activbb5==1) {fault++; break ;}
10    // body execution
11 // Mandatory mutation on bb1: bb2/bb3
12        if(buffer[i] != pin[i]) fault++;
13        i++ ;
14  }
15 // Comparison is successful: this block is always reached.
16  triesLeft = maxTries;
17  authenticated = 1;
18  return EXIT_SUCCESS;
19 }
```

Listing 4. Mutant VerifyMutComp.ll stated in C

IV. Fault injections and Robustness evaluation

Starting from a mutant we want to determine if some attacks exist, with how many fault injections and where. Due to the chosen approach, where a single mutant is produced including all possible fault injections, robustness can be evaluated using a test generation process targeting a path coverage criterion. Subsection IV-A explains how the test campaign is conducted and Subsection IV-B how this result is exploited to evaluate the robustness against fault injections.

A. Symbolic test generation

In order to combine at best inputs and fault injections, we use a symbolic (or concolic) test case generation approach. Our choice is bearing on KLEE [13], [4], a free symbolic test generation tool for LLVM[6]. In KLEE, by default, variables are considered as concrete ones. They can be declared as symbolic (using the function `klee_make_symbolic`). Furthermore, we can state assertions using the predefined function `klee_assume(pred)`, `pred` being a C condition. First we build drivers for test generation in the following way:

1) add assertions characterising the inputs we want to take into account ;
2) put an assertion to successively compute paths satisfying the following conditions: `fault==0`, `fault==1`, `fault==2` and `fault>=3`.

The case `fault==0` corresponds to the case when the goal can be reached without any attack using chosen inputs. In this case there is a problem somewhere (either in the goal, or in the chosen inputs, or in the application itself). Cases `fault==1` and `fault==2` give attack scenarios, according to the current state of the art. In absence of such attacks, the last case gives a measure of robustness: the minimal value of required fault injections for the fault model under consideration. We give in Listing 5 a KLEE driver associated to the function `Verify`, where `triesleft` remains concrete (equals to 3) and `buffer` is declared as symbolic (line 9) with each byte constrained to be different from the expected pin value (line 13). Furthermore, paths with two fault injections are targeted (line 15).

```
1 #define SIZE_OF_PIN 4
2 #define maxTries 3
3 typedef unsigned char BYTE;
4 BYTE triesLeft = maxTries;
5 BYTE authenticated = 0;
6 BYTE pin[4] = {(char)1, (char)2, (char)3, (char)4};
7 int fault = 0;
8 int main(void) {
9     klee_make_symbolic(buffer, sizeof(BYTE)*SIZE_OF_PIN,
10                        "buffer");
11 //Conditions on the input buffer
12     for (i=0 ; i<SIZE_OF_PIN ; i++)
13         {klee_assume(buffer[i] != pin[i]); }
14     Verify(buffer);
15     klee_assume(fault == 2);
16 }
```

Listing 5. A KLEE driver for function Verify

[6]Previous experiments have been made at the C code level using the Pathcrawler tool [17], [18]

From this driver addressing the mutation of the function `Verify` (Listing 4), KLEE produces inputs for all feasible paths, according to the stated assertions. In particular all paths introduced by our mutation scheme are explored (variables `activX` being successively assigned with 0 and 1).

B. Robustness evaluation

Symbolic test generation tools may face two forms of incompleteness: unbounded paths enumeration and undecidability of constraints solving. Termination is then enforced using timeouts. For a given driver, containing an assertion A_1 on inputs and an assertion A_2 stating a fault limit, robustness evaluation is established according to table III.

Attack	at least one executable paths ensuring $A_1 \wedge A_2$
Inconclusive	a timeout detection and no attack produced
Robust	no timeout and no attack

TABLE III
Possible Verdicts attached to a given driver

These verdicts have been implemented using KLEE. This tool produces one test case for each enumerated path respecting the assertions and a single test case violating each assertion. It interacts with the solver STP [19]. By default, KLEE can loop when paths (or number of paths) are infinite. We developed a script adjusting KLEE parameters in order to detect a timeout either during paths enumeration, or due to STP limits.

For each produced test case, KLEE builds a file named **testX.ktest** (X being the test case number) containing the chosen values for symbolic variables activing a given path. Symbolic inputs can be displayed using the KLEE's *ktest-tool* command. Additionally, KLEE produces a file `.early` when a timeout is raised and a file called `testX.user.err` for each test case `X` violating an assertion. Thus, we can conclude on the robustness of an application depending on the presence or the absence of these files, as shown Table IV.

Attack	.ktest files without a .user.err associated file
Inconclusive	a .early file without attack
Robust	no .early file and every .ktest file associated with a .user.err file

TABLE IV
KLEE Verdicts attached to a given driver

As pointed out in section III-A, the mutation strategy Comp succeeds (i.e., reaches a block or not), only if the execution terminates normally. Then, some executions can diverge (looping forever or with an unpredictable behaviour due to an execution error). Effect of execution errors, due to an erroneous program or provoked by a fault injection, is not addressed here. Nevertheless, thanks to KLEE, we can identify attacks raising an execution error: KLEE produces a particular error file depending on the error type, for instance a `testX.ptr.err` file for a pointer error. Such attacks are met in the example of section V-C.

Number of fault injections	without any constraints on inputs	with conditioned inputs
fault ==0	1	0
fault ==1	x	**1**
fault ==2	x	**1**
fault >= 3	x	**3**

TABLE V
NUMBER OF POSSIBLE ATTACKS FOR VERIFY

Number of	All detected attacks	With preorder filtering
fault ==0	0	0
fault ==1	**1**	**1**
fault ==2	**1**	0
fault >= 3	**3**	**1**

TABLE VI
ATTACKS FOR VERIFY AFTER SELECTION

C. Robustness evaluation for the function Verify

We make two experiments using the mutant `VerifyMutComp.ll`. Results are presented Table V.

In the first experiment (column 2 of Table V) we use a driver targeting 0 fault without any constraints on inputs, in order to verify for which inputs the implementation is unsecure (here when the attacker knows the right pin).

During the second experiment (column 3 of Table V) we produce 4 drivers, instantiating the `klee_assume` with the expected values of `fault` (0, 1, 2 and ≥ 3) and conditioned inputs as stated on Listing 5. KLEE obviously supplies an exhaustive coverage of all paths in a very negligible time. This naive implementation of `Verify` presents one obvious 1-shot weakness (see Example 4.1). The 2-shots attack consists in forcing the internal condition `buffer[1]!=pin[1]` and then exiting the loop. Attacks with `fault >=3` correspond to forcing 2 or 3 times the internal condition and then the loop exit, or to forcing 4 times the internal condition (example 4.2).

Example 4.1: A 1-shot attack scenario.

The attack column 3 line 3 consists in circumventing the loop execution by the following sequence of fault injection: `<bb4:bb1/bb5>`.

Example 4.2: A 4-shots attack scenario.

One possible 4-shots attack consists in the sequence of fault injections `<bb1:bb2/bb3, bb1:bb2/bb3, bb1:bb2/bb3, bb1:bb2/bb3>`, meaning that a fault is injected four times on node `bb1` to hijack the flow from `bb2` to `bb3`. Input value produced by KLEE for `buffer` is 0000.

V. EXPERIMENTS

A. Filtering Attacks

When dealing with non trivial examples, the number of attacks found may combinatorially increase with the number of injected faults, in particular due to the fact that we consider volatile fault injections that can occur several times or at different locations of the execution. An obvious strategy consists in first examining attacks with the smaller number of fault injections. We propose here another complementary criterion, taking into account program locations where faults should be injected (physical realization of a given fault injection could be more or less practicable). For a given attack a, seen as an execution trace, we denote as $\text{Faults}(a)$ the *multi-set* of program locations where faults have been effectively injected. Definition 5.1 defines a preorder relation between attacks.

Definition 5.1: A preorder relation $\preceq$ between attacks.

$$a_1 \preceq a_2 \;\equiv\; \text{Faults}(a_1) \subseteq \text{Faults}(a_2) \quad \text{(multi-set inclusion)}$$

Due to the fact that some attacks could diverge, we apply this order only on attacks terminating normally. In this case we first select the minimal elements with respect to the relation $\preceq$. We give in Table VI the subset of attacks that fulfils our selection criterion for the function `Verify`. Only attacks of examples 4.1 and 4.2 are selected. Unselected attacks consist in mutating the internal condition one, two or three times and then forcing the loop exit: they are redundant with attack 4.1 in which the loop exit is directly forced. They include the set of fault injection locations of attack 4.1, plus some other ones.

The proposed filtering criterion should be easely adapted to take into account the difficulty of fault injection, depending on the program location where it takes place. In practice, one difficulty is to control physical fault injection in time, i.e, when a given instruction is in progress (ongoing). To do that, execution traces are analyzed, using observable side channels such as power traces (for instance a peak due to a cryptographic calculus) [9]. Therefore, injecting a fault at a given instruction also depends on the preceding instruction. Definition 5.1 should be refined in order to take into account observable execution locations.

B. The Lazart Tool Suite

The cfg coloring process is implemented in a Java standalone tool, according to section II, and is no more detailed here. The mutation production and robustness evaluation processes consist in the following steps:

1) LLVM compilation (using `llvm-gcc` or `clang`)
2) Mutation generation (the Wolverine tool)
3) Symbolic analysis (using KLEE)
4) Attack filtering (an home-made script)

The structure of this tool suite is described Fig. 6. The *main* and the *target* modules are LLVM bytecode files respectively containing the *main* and the function targeted by the fault injection. The *fault model* and the *mutation strategy* precise which faults should be injected and where (the name of the targeted function and the set of mutations to apply on each block). Optionally a faults number limit l specifies a stopping criterion for the symbolic analysis. We now describe the tasks flow of the Lazart tool suite.

1) Mutant generation: Wolverine takes a mutation strategy as input and automatically modifies the LLVM bytecode in order to apply the corresponding mutation patterns. It also produces a KLEE compatible driver. Wolverine was built to be extensible to some other types of fault, through its strategy parameter.

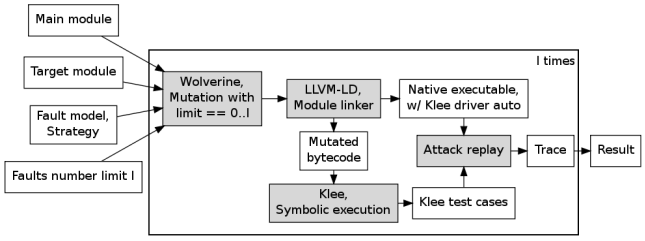

Fig. 6. The tool structure

2) Robustness evaluation: Then we use *LLVM-LD*, a linker, which allows to combine several LLVM modules in a single application. The resulting file is given to KLEE which produces test cases as described section IV.

3) Running tests: Test cases obtained from KLEE can be executed using the native code supplied by *LLVM-LD*, a `testX.kest` file and the `libkleeRuntest` library, linking symbolic variables with their effective values.

4) Attack filter criterion: Attack execution traces are used to implement the preorder relation between attacks (see section V-A). Wolverine puts a `printf` call in the code each time a fault is injected in order to log the source and the target blocks of the control flow hijack. Then, after their execution, attacks can be compared and classified.

C. Taking into account countermeasures

We now apply the Lazart approach on a more secured implementation of the function `Verify` (inspired from [14] and given Listing 6) in order to evaluate how countermeasures are taken into account. Countermeasures introduced here are the following ones:

- a backup of `triesLeft` is introduced (`triesLeftBackup`) and compared with a copy of `triesLeft` (the local variable `t1` that will be stored in RAM). If this copy and the backup value differ an attack is detected (lines 11, 16 and 28)
- a step counter ensuring that the loop is really executed 4 times (incremented into the loop and tested at the end of block `bb12`, line 31)
- a redundant test on the value of `equal` after the loop, in order to detect a test inversion (blocks `bb9` and `bb10`, lines 23 and 25).

We now combine two attack objectives: in Fig. 7(a) we want to reach `bb12` (forcing the execution of `authenticated=1` as before) and in Fig. 7(b) we want not to reach the block `CM` (corresponding to the detection of an attack by a countermeasure). We obtain the following mutation directives:

```
Mandatory mutations:
  entry: CM/bb, bb: bb1/bb2,
  bb2: CM/bb3, <bb9: bb15/bb10,
  bb10: CM/bb11, bb11: CM/bb12,
  bb12: bb14/bb13
Optional mutations:
  bb8: bb4/bb9, bb8: bb9/bb4,
  bb4: bb5/bb6, bb4: bb6/bb5
```

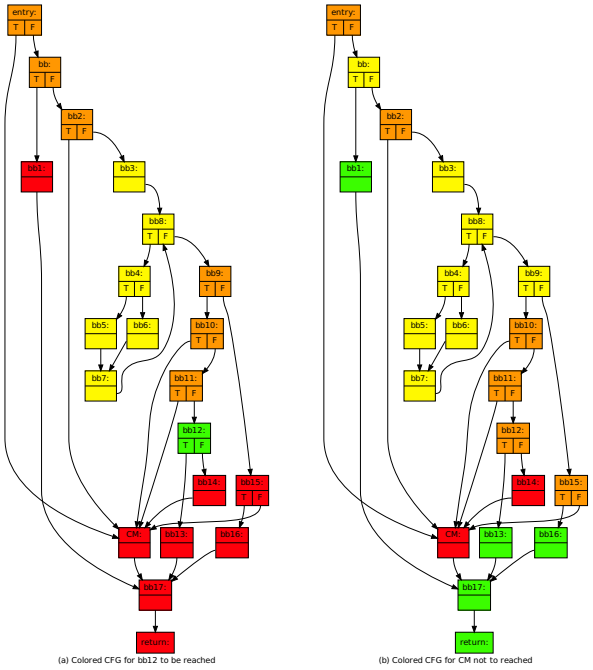

Fig. 7. Colored CFGs with goals bb12 and CM

Considering conditioned inputs as before (see Listing 5), Table VII shows that we obtain a path leading to an error when we force the execution of the loop after 4 steps (an out of bound error is detected by KLEE). This path is denoted by 1^* in Table VII and requires to be examined as a potential attack, depending on the execution platform. We now have two 2-shots attacks (line 4 of Table VII). The first one corresponds to the injection of faults successively on the two conditions `equal==BOOL_TRUE` (`bb9`) and `equal!=BOOL_TRUE` (`bb10`). The second one corresponds to the the 1-shot attack of example 4.1 that now requires two shots: one to circumvent the loop execution and another one to hijack the countermeasure relative to the the step counter value (line 31).

Number of	All detected attacks	With preorder filtering
fault ==0	0	0
fault ==1	1^*	1^*
fault ==2	2	2
fault ==3	5	0
fault ==4	11	1
fault >=5	13	0

TABLE VII
RESULTS FOR SECURED VERIFY

This example also shows that we can relate countermeasures and fault injections. Countermeasures can be seen as blocks directly leading to a state detecting an attack (here the block CM). For instance the orange nodes of Fig. 7(b), i.e.

```
1 signed char triesLeft = maxTries;
2 signed char triesLeftBackup = -maxTries;
3 BYTE  equal = BOOL_TRUE;
4 BYTE authenticated = 0;
5 BYTE pin[4]={(char)0,(char)1,char(2),char(3)};
6 BYTE Verify(char buffer[4]) {
7    int i;
8 //BLOCK entry
9    int stepCounter = INITIAL_VALUE;
10   short char t1 = triesLeft;
11   if(t1 != -triesLeftBackup) goto CM ;
12 //BLOCK bb
13   if(triesLeft <= 0) return EXIT_FAILURE;
14 //BLOCK bb2
15   t1--; triesLeft = t1; triesLeftBackup++;
16   if(triesLeft != -triesLeftBackup) goto CM ;
17 //BLOCK bb3
18   equal = BOOL_TRUE;
19   for(i = 0; i < 4; i++)
20      {equal=equal&((buffer[i]!=pin[i])?BOOL_FALSE:
           BOOL_TRUE);
21       stepCounter++; };
22 //BLOCK bb9
23   if(equal == BOOL_TRUE) {
24 //BLOCK 10
25      if(equal != BOOL_TRUE) goto CM ;
26 //BLOCK bb11
27      triesLeft = maxTries; triesLeftBackup = -maxTries;
28      if (triesLeft != -triesLeftBackup) goto CM ;
29 //BLOCK bb12
30      authenticated = 1;
31      if(stepCounter == INITIAL_VALUE + 4)
32 //BLOCK bb13
33         return EXIT_SUCCESS; }
34 //BLOCK bb14 : else part of  block bb12
35 //             followed by block CM
36   else { // Comparison failed
37      authenticated = 0;
38      if(stepCounter == INITIAL_VALUE + 4)
39 //BLOCK bb16
40         return EXIT_FAILURE; }
41 // BLOCK  CM
42       CM : printf("Card blocked.\n");
43       return EXIT_FAILURE;
44 }
```

Listing 6
A SECURED IMPLEMENTATION OF VERIFY

`entry`, `bb2`, `bb10`, `bb11`, `bb12`, reaching block `CM` can be identified as containing countermeasures. Generated attacks only imply nodes `bb10` and `bb12`. We can conclude that countermeasures in `entry`, `bb2` and `bb11` are not related to fault injections by test inversion for the considered inputs (`TryLeft` being set to 3) . Thus, using reachability or unreachability objectives, we are able to determine if a given countermeasure can be raised and for which inputs.

VI. CONCLUSION

A. Our Contribution

In the context of smart card application certification, we propose an innovative approach allowing us to establish commensurable verdicts (existence or absence of attacks relatively to a set of possible inputs), unlike other approaches using an empirical testing phase to check if some mutants can be exercised by some inputs, and if such executions are really dangerous ones (i.e. not detected by countermeasures). Furthermore, as shown in section V-C, we are able to evaluate how embedded countermeasures are related to attack detections. To the best of our knowledge, we are the first to built mutant taking into consideration multiple shots fault injection during a run, according to the next state-of-the art in term of laser attacks.

The proposed approach is supported by an effective tool suite, as described section V-B4. It is based on a robust and efficient symbolic test generator, used in industrial contexts [20], [21]. LLVM appears to be a suitable code level both to produce mutants and to take into account optimisations, thanks to the analysis and transform passes offered by the LLVM platform [22]. Thus, several level of optimised codes can be considered, bridging the gap between high level attacks and effective attacks, made at the binary level.

Regarding mutation, as pointed out in [23], [24], combining mutation production and symbolic test generation offers several advantages. Firstly, we conjointly address the problem of building mutants and how they can be activated. Secondly, thanks to the efficiency of the KLEE path-coverage strategy, we are able to master the combinatorial explosion inherent to multiple and volatile fault injections, in combining at best inputs and faults injection and in pruning as soon as possible paths violating stated assertions relative to the expected number of faults.

To complete the evaluation of the Lazart approach, we experimented it on a cryptographic detector for ssh, executing some checks allowing to verify the integrity of packets (see [25] for attacks on ssh packets). We tested the `detect_attack` function provided in OpenSSH 6.2 [26] (70 lines of C code and a cfg with 32 basic blocks). Our objective was to hijack the function verdict (i.e, to report as correct a corrupted packet). The cfg coloring step produced 15 mutations. We ran Klee looking for possible attacks in less than 3 fault injections, with a symbolic 1024 size buffer. We found a 1-shot attack (inverting a check about the packet length), and a 3-shots attack (redundant with the previous one, according to definition 5.1). Although the code structure is similar to the one of the `Verify` example, this example involves more complex arithmetic computations, a large symbolic buffer and much larger iteration depths. However, the results were produced in a few minutes.

B. Related work

Fault injection by laser attacks is a hot topic in the domain of smartcard vulnerability analysis (and generally mobile devices). Thus, several recent works are related to this subject.

For instance Lanet et al. [15], [11] produce mutants for smart card fault injection for Java card applications. Based on a fault model at the byte level [5], they produce a set of mutants by mutating one byte code operation per mutant. This way, they encompass a large set of fault injection impacts, for one shot laser attacks. For dangerousness detection, they propose a set of syntactic heuristics to conduct a risk analysis for each mutant. Furthermore, they use the annotation facility offered by Java Card 3.0 to annote applications with declared countermeasures. Their impacts are then measured by a symbolic execution interpreting annotations. This approach mainly targets Java Card developers in offering a framework to evaluate countermeasures accuracy, according to a given fault

model. It differs from Lazart in several points: only single permanent fault injections are considered and countermeasures must be explicitly declared. On the contrary they consider a more general fault model.

In [7] Berthomé et al. produce C mutants corresponding to a single permanent fault injection, also targeting control flow hijacking. There are some similarities with our approach, in the sense they try to bridge the gap between the assembly code and the source code by modelling and simulating the effect of physical attacks directly at high level. One interesting aspect of [7], [12] is the systematic study of attacks targeting control flow shifting and how mutant for NOP and JUMP attacks can be systematically produced. Mutants are to be used against a low level attack platform and predefined test scenarii in order to verify that abstract mutants are covered by concrete fault injections. In contrast, Lazart approach aims to exploit abstract models to produce a robustness verdict and existing attacks.

In a very recent work [8], [27], the author focus on fault injection consisting in modifying memory values (a classical attack for cryptographic algorithms), aiming to formally prove the robustness of a given application. Starting from a C code, the author produce mutants for a single fault injection, adding an extra parameter similar to the one introduced in our optional mutation operator. This parameter is used to logically express a robustness property stating that if a fault has been introduced, the program stops in error (a countermeasure state for instance). Robustness property is established using the weakest precondition plugin Jessy of the Frama-C platform [28], [29]. The use of a symbolic test case generator in Lazart, contrary to a theorem-proving approach, allows us either to establish the robustness or to produce attacks. Furthermore, mixing concrete and symbolic computation steps allows us to automatically treat applications with complex arithmetic computation (such as shift and hash function), as met in the ssh experimentation. It is not sure that such applications can be easely treated by a theorem-proving approach.

C. Perspectives

In parallel with experimentations, we are currently developing this work following two main axes.

First we are studying how to extend our approach to other forms of attacks on control flow hijacking such as circumventing or forcing some calls, or forcing some block execution depending on the code layout. This attack objective can be realized, for instance, in injecting NOP instruction or in changing the destination address of JUMP, as proposed in [7]. For these attack objectives, the Lazart approach appears to be extensible in adding or modifying some edges in the control flow graphs. We are currently extending the coloring algorithm to take into account inter-procedural analysis.

Secondly we are studying how to combine some aspects of the Lazart approach (in particular graph coloring and mutation strategies) with a (brute force) dynamic testing approach at the byte level, in order to guide fault injections at execution time. Nevertheless, static analysis and cfg production are more challenging when low level codes are concerned [30].

REFERENCES

[1] "Application of Attack Potential to Smartcards," Commun Criteria, Tech. Rep. CCDB-2009-03-001, march 2009.

[2] "Common Criteria for Information Technology Security Evaluation, Part 3: Security assurance components," Tech. Rep. CCMB-2006-09-003, v3.1, sept 2006.

[3] T. L. C. Infrastructure, http://llvm.org.

[4] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[5] J. Blömer, M. Otto, and J.-P. Seifert, "A new CRT-RSA algorithm secure against bellcore attacks," in *CCS '03*. New York, NY, USA: ACM, 2003, pp. 311–320. [Online]. Available: http://doi.acm.org/10.1145/948109.948151

[6] I. Verbauwhede, D. Karaklajic, and J. Schmidt, "The fault attack jungle - a classification model to guide you," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, 2011, pp. 3–8.

[7] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky, and J. Lalande, "High level model of control flow attacks for smart card functional security," in *ARES 2012*, aug. 2012, pp. 224 –229.

[8] M. Christofi, "Preuves de sécurité outillées d'implémentation cryptographiques," Ph.D. dissertation, Laboratoire PRiSM, Université de Versailles Saint Quentin-en-Yvelines, France, 2013.

[9] H. B.-E. Hamid, H. Choukri, D. N. M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," 2004.

[10] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[11] J.-B. Machemie, C. Mazin, J.-L. Lanet, and J. Cartigny, "SmartCM a smart card fault injection simulator," in *IEEE International Workshop on Information Forensics and Security*, 2011.

[12] X. Kauffmann-Tourkestansky, "Analyses sécuritaires de code de carte à puce sous attaques physiques simulées," Ph.D. dissertation, Université d'Orléans, 2012.

[13] "The KLEE symbolic virtual machine," http://klee.llvm.org/.

[14] V. Eric, "JC101-12C: Defending against attacks," javacard.vetilles.com/2008/05/15/jc101-12c-defending-against-attacks/.

[15] A. Sere, J.-L. Lanet, and J. Iguchi-Cartigny, "Evaluation of Countermeasures Against Fault Attacks on Smart Cards," *International Journal of Security and Its Applications*, vol. 5, no. 2, pp. 49–61, 2011.

[16] Y. Jia and M. Harman, "Higher order mutation testing," *Journal of Information and Software Technology*, vol. 18, no. 3, pp. 1379–1393, 2009.

[17] CEA-LIST, "Test your code," pathcrawler-online.com.

[18] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams, "Automating structural testing of c programs: Experience with pathcrawler," in *AST '09*, may 2009.

[19] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV*, 2007, pp. 519–531.

[20] Y. Kim, M. Kim, Y. J. Kim, and Y. Jang, "Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE," in *Software Engineering (ICSE), 2012 34th International Conference on*, 2012, pp. 1143–1152.

[21] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408795

[22] L. Analysis and T. Passes, http://llvm.org/docs/Passes.html.

[23] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *ISSRE*, 2010, pp. 121–130.

[24] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *SIGSOFT FSE*, 2011, pp. 212–222.

[25] C. Security, http://www.coresecurity.com/content/ssh-insertion-attack.

[26] OpenSSH, http://www.openssh.org.

[27] M. Christofi, B. Chetali, L. Goubin, and D. Vigilant, "Formal verification of a crt-rsa implementation against fault attacks," *J. Cryptographic Engineering*, vol. 3, no. 3, pp. 157–167, 2013.

[28] "Frama-C, software Analyzers," http://frama-c.com.

[29] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c - a software analysis perspective," in *SEFM*, 2012, pp. 233–247.

[30] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems*, vol. 32, pp. 23:1–23:84, August 2010.