An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries

Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, Axel Legay

▶ **To cite this version:**

Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, Axel Legay. An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries. 2017. hal-01400283v1

HAL Id: hal-01400283

https://inria.hal.science/hal-01400283v1

Preprint submitted on 30 Jan 2017 (v1), last revised 4 Apr 2017 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries

Thomas Given-Wilson
Inria
Rennes, France
thomas.given-wilson@inria.fr

Nisrine Jafri
Inria
Rennes, France
nisrine.jafri@inria.fr

Jean-Louis Lanet
Inria
Rennes, France
jean-louis.lanet@inria.fr

Axel Legay
Inria
Rennes, France
axel.legay@inria.fr

ABSTRACT

Recently fault injection has increasingly been used both to attack software applications, and to test system robustness. Detecting fault injection vulnerabilities has been approached with a variety of methods, yielding varied results. This paper proposes a general process using model checking to detect fault injection vulnerabilities in binaries. The process is implemented and used to detect a variety of different kinds of fault injection vulnerabilities in binaries.

CCS Concepts

•**Security and privacy** → **Hardware attacks and countermeasures;** *Embedded systems security; Software and application security;* •**Software and its engineering** → **Software verification and validation;** *Model checking;*

Keywords

Fault Injection; Model Checking; Fault Tolerance; Vulnerability Analysis; Dependability; Security

1. INTRODUCTION

Recently fault injection has been increasingly used both as a method to attack software applications, and to test the robustness of software systems. Many systems are particularly vulnerable to fault injection attacks due to operating in hostile environments, i.e. environments where an attacker may be able to perform physical attacks on the system hardware. Many such attacks have been demonstrated on a variety of systems, showing that different kinds of faults can be injected into various devices [4, 15, 45].

The wide variety of fault injection attacks and possible impacts upon a system make it impossible to prevent software from failing under all possible attacks [45]. Thus, recent work has approached the problem of fault injection by limiting the scope of attacks, or limiting the kinds of vulnerabilities analysed [5, 28, 11, 29]. This paper proposes a formal process for the detection of fault injection vulnerabilities in binaries. In particular, a process that can account for many different kinds of fault injections and that does not require extensive hardware or specialised equipment. This process is achieved by simulating fault injection attacks upon the executable binary for the given software, and then using model checking to determine whether or not the simulated fault injection attack violates properties the software should maintain. Thus, a violation of the properties indicates fault injection vulnerability that may be exploitable by an attacker.

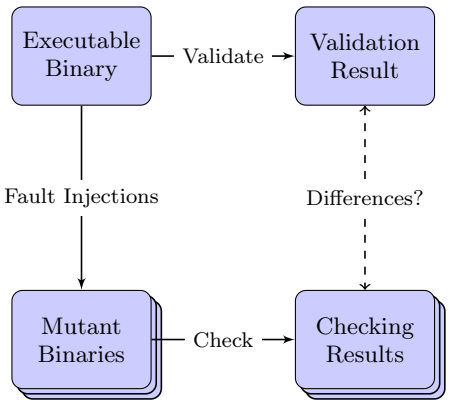

Figure 1: Process Overview

This paper presents a process for detecting vulnerabilities in binaries using model checking, as summarised in Figure 1. The process begins with the *executable binary* that represents the program to be considered. The validation of the binary involves checking various *properties* using model checking to ensure the binary meets it's specification. A fault injection attack is simulated on the executable binary, producing a *mutant binary*. The properties are then model checked on the mutant binary. A difference in the result between validating and checking the properties indicates a vulnerability to the fault injection attack that was simu-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASIACCS 2017, Abu Dhabi, UAE

© 2017 ACM. ISBN isbn-code-number...$15.00

DOI: doi.number

lated.

This process provides a general approach that can support detecting a wide variety of fault injection vulnerabilities in binaries. Several key points include the following. By operating directly upon the binary, fault injection vulnerabilities that cannot be detected in source languages or intermediate representations can be detected [37]. Formal methods, here model checking, ensure the rigour of the analysis and so ensure that fault injection vulnerabilities that are detected are real and not false positives. By designing a process that is easy to iterate over various fault injection models and approaches, analysis of fault injection vulnerability can be easily automated. An automated process can be implemented in a manner that allows broad, or even complete, coverage of possible fault injection attacks. Combining automation, broad coverage, and formal methods, allows the process to make strong guarantees about the vulnerability of a system that has been analysed.

To support the feasibility of implementing the process in a manner that could be easily automated, this paper also presents an example implementation. The choice of implementation here is to use currently available tools and demonstrate the feasibility, rather than generality, of implementation. Indeed there are many possible ways to implement this process. The choice of tools to use depends significantly upon the target architecture of the executable binary, and the scope of the properties that are to be checked. Although compiling to many different architectures is straightforward, model checking executable binaries is more complex [6]. In practice directly model checking a binary executable is infeasible, and so the usual approach is to convert the executable binary into an intermediate representation to be used by the model checker [21, 22]. The choice of intermediate representation can be limited both: by the executable binary form, such as being a specific architecture; or by the choice of model checker if fixed a priori. The choice of model checker can be quite significant, since many have limitations on the properties they can check, and some of these also depend on the intermediate representation [34, 26, 9, 41].

Thus, the implementation presented here uses currently available tools that easily fit together. Compilation is handled directly by GCC [14]. The validation and checking are done by a combination of MC-Sema [44] (to convert executable binaries into an intermediate representation) and LLBMC [40] (to model check the intermediate representation). Although adequate for a proof of concept, both MC-Sema and LLBMC have limitations that make them inadequate to be a general implementation of the process. In particular, MC-Sema is limited to X86 architecture, and is unable to convert all X86 instructions into an intermediate representation [44], thus MC-Sema is unsuitable for a general solution. LLBMC can only support limited properties to be model checked, which is again unsuitable for a general solution to the process. Despite these limitations in the tools, the implementation is able to illustrate the feasibility, versatility, and strength of the process.

To demonstrate the versatility and strength of the process, experimental results are presented that use the implementation described above. These results are presented to illustrate detecting several fault injection vulnerabilities on an executable binary of a simple PIN verification program. The experimental results cover two areas.

The first collection of experiment results are built up illustrating the significance of choosing correct properties to be checked, and illustrating different kinds of fault injection vulnerabilities. Once appropriate properties have been established, several different kinds of fault injection attack are demonstrated, covering a variety of attack models [17, 4]. This includes attack models that only exist upon the executable binary and cannot be detected either upon the source code alone, or by "compiling" to an intermediate representation. These results demonstrate the ability to detect various fault injection vulnerabilities on an executable binary. In particular, they demonstrate how to do this using existing tools and formal methods, in a manner that can be applied to any binary without requiring specialised hardware or expensive resources.

The second collection of experimental results demonstrate the feasibility of automating the process and implementation. A simple fault injection tool is created, and used to automate the detection of fault injection vulnerabilities in a simple executable binary. A script is then used to automate fault injection vulnerability detection, and experiments run to determine the runtime and number of fault injection vulnerabilities found. These results show that the process is feasible to automate, even with the implementation presented here and without optimisation.

The key contributions of this paper are as follows.

- Describing a general process that allows automated detection of fault injection vulnerabilities in binaries.
- Detailing an implementation of the process that allows easy automation with existing tools.
- Illustration of the process and implementation with a clear example that highlights the feasibility of the above.
- Experimental results showing that the process can be automated, and further that running the experiments is computationally feasible.

The structure of the paper is as follows. Section 2 presents a simple motivating example that is used to clearly demonstrate the techniques. Section 3 recalls background on fault injection attacks, model checking, and properties. Section 4 details the process proposed in this paper. Section 5 discusses the implementation and tools used to achieve the process. Section 6 presents experimental results for finding fault injection vulnerabilities in binaries. Section 7 discusses related work. Section 8 concludes and discusses future work.

2. MOTIVATING EXAMPLE

This section presents a motivating example that is used to: illustrate the concepts and process, and for the experimental results of this paper. The example is of code that checks a PIN supplied by a user when authenticating to use a credit card.

Consider the following code that checks the value of a candidate PIN entered by a user when authenticating to use a credit card. Prior to this code fragment the true PIN `PINTrue` is assumed to be defined and intialised with the true PIN value. Similarly the candidate PIN `PINCandidate` is defined and initialised with a value input by the user. Further, both PINs are checked to be the same length and this length is defined to be their size `PINSize`.

```
1  bool grantAccess = false;
2  bool badValue = false;
3  int i = 0;
4  while (i < PINSize) {
5    if (PINCandidate[i] != PINTrue[i]) {
6      badValue = true;
7    }
8    i++;
9  }
10 if (badValue == false) {
11   grantAccess = true;
12 }
```

The first three lines initialise variables to be used in the code fragment shown here. The `grantAccess` variable is used to indicate whether or not to grant access after checking the candidate PIN against the true PIN, intialised to `false`. The `badValue` variable is used to detect when digits of the two PINs do not match, also intialised to `false`. The variable `i` is used as a an iterator to progress through the digits of the PINs, intialised to `0`. The next six lines of the code are a loop that iterates through the digits of the candidate PIN and true PIN, incrementing `i` on line eight, and bounded by the PIN size `PINSize`. Line five checks the `i`th digit of the candidate PIN against the `i`th digit of the true PIN. On line six, if the digits differ, then `badValue` is set to `true` to indicate that (at least one) digit of the two PINs are not equal. At the end, on line ten the conditional checks that no bad values have been found, and grants access if this is the case.

Now consider when `PINSize = 4`. By changing a single bit, an attacker could change the value of `PINSize` from `4` to `0`. (This succeeds since `4 = 0...0100` in binary, and changing the `1` to `0` yields `0...0000`.) Observe that this would bypass the loop since `i < PINSize` (i.e. `0 < 0`) would not hold, and therefore the checking of any digits of the candidate PIN. Thus, the example is vulnerable to this kind of 1-bit fault injection attack (as well as several others that will be introduced later).

The above paragraph describes a fault that can be injected into the executable binary that would allow the attacker to gain access even without the correct PIN. This paper proposes and explains a process that can be used to detect such fault injection vulnerabilities (Section 4). An implementation of the process is detailed in Section 5, and this implementation is used to obtain the experimental results (Section 6). The motivating example is used as the basis for the experimental results, including detecting the fault injection vulnerability described above.

3. BACKGROUND

This section recalls key concepts useful to understanding the rest of the paper. These are divided into three main areas: fault injection, model checking, and properties.

3.1 Fault Injection

Fault injection can be considered as an attack, when an attacker targets the hardware of a system to create an exploitable error at the software level. The goal of such an attack is to cause a specific effect at the hardware level, that in turn creates an exploitable change in the software behaviour. The rest of this section discusses classification of, and different kinds of, fault injection attacks.

The hardware effect of a fault injection attack is described through a *fault model* that specifies the nature and scope of the induced modification. Typically such attacks are achieved by changing a value stored in the hardware, such as switching the value of a single bit, or changing the value of a whole byte [45]. Such hardware effects generally focus on the kind of fault that can be created rather than the effect this has on the software.

The effects of faults can be classified into three categories: *data flow* [2, 16], *control flow* [27], and *instruction modification* [38]. Data flow faults affect the values of data used within a program. A couple of examples include: register corruption, that changes the value stored in a register; or memory corruption, that changes the value stored in memory. More generally, data flow considers all faults that modify data values in registers, memory, or values used in instructions. Control flow faults effect the order in which instructions are executed, i.e. the general control flow of execution. Examples include: modifying the target of a jump instruction, or by skipping instructions. Instruction modification faults target a code instruction in some specific manner. Examples include: changing an ADD instruction into a SUB instruction; or rewriting the whole instruction to be a NOP (non-operation). Observe that instruction modification faults can be considered as a superset of data flow and control flow faults when working with executable binaries. This is since data flow and control flow are both achieved via machine instructions in the executable binary, and so data flow and control flow faults can be achieved via instruction modification faults in practice on executable binaries. Therefore, the rest of this paper will not distinguish between different kinds of faults, instead treating them all as instruction modification faults.

Simulating fault injection attacks in an experimental environment can be done in two ways: reproducing the attack on the hardware, or simulating the attack with software. Reproducing the attack using hardware technology is relatively difficult and expensive, since specialised hardware must be used to inject the fault (e.g. using a laser [5], or electromagnetic pulse [28]). Comparatively, simulating with software is easy and cheap because this requires only modification to the executable binary and no specialised hardware. Since the goal in this paper is to develop an efficient process that can be implemented with a software tool chain, the rest of this paper will only consider software simulations.

Software simulation attacks can also be classified into two kinds of fault injection attacks, *run time* and *compile time*. Run time fault injection attacks are those that occur only while the code being attacked is being executed. Compile time fault injection attacks are those that occur at any time starting form compilation of the code, and up until just prior to execution. This paper considers only compile time fault injection attacks since this builds towards future work (see Section 8.2).

3.2 Model checking

Model checking is a formal method for determining whether properties hold on a model [3]. Model checking has the advantage that all possible states of the model are considered, and so is guaranteed to be able to answer whether or not a property holds for a given model.

The *model* is a representation of the program or system

being considered. A good model is able to represent all the possible states and transitions that the program can achieve. In this work, the model represents an executable binary program.

The cost of model checking comes in the potential exponential complexity used to consider all the possibly infinite states of a model. However, for limited models, model checking is highly efficient and precise. Further, various approaches have been used to make model checking efficient even for large and complex models (or programs) [20].

Bounded model checking is a refinement of model checking that alleviates some of the issues with possibly infinite complexity by bounding the checking [10]. The key idea in bounded model checking is to put a bound on parts of the model that could be infinite (or at least extremely large). For example, checking a program with a loop, going through hundreds or millions of iterations could be very costly for model checking. However, bounded model checking of such an example could limit the number of times to iterate through a loop. Thus, bounded model checking allows limits to be placed upon such potentially unbounded aspects of model checking.

3.3 Properties

To perform model checking requires specifying the *properties* to be checked upon the model. There are two main kinds of properties that can be checked *safety*, and *liveness* [3]. Safety properties are used to express that certain propositions hold when they are encountered. Liveness properties express that propositions hold over some temporal dimension. This paper only considers safety properties since these are clearer, more intuitive to represent, and sufficient to illustrate the feasibility of the process. Liveness properties can also be checked in a similar manner, although this is not presented in this paper, for further details see the discussion in Section 8.2.

Safety properties can be expressed by simple propositions that can be annotated into the code of the program being considered. Recalling the motivating example (in Section 2), a naive safety property could be expressed by an assert statement such as

```
__llbmc_assert(i == 4);
```

that is inserted between lines 9 and 10. This property would be checked by the model checker to ensure that the variable `i` has the value `4` at this point in the model.

More generally such asserts support properties defined with boolean propositions. Here we exploit properties supporting: negation denoted `!`, equality denoted `==`, inequality denoted `!=`, conjunction denoted `&&`, and disjunction denoted `||`. For example, the following property

```
__llbmc_assert(    !(PINCandidate != PINTrue)
                || grantAccess == false);
```

combines negation with inequality, disjunction, and equality. The semantics are that when the two PINs are not equal, then access is not granted.

4. PROCESS

This section details the process presented in this paper for detecting fault injection vulnerability in binaries.

An overview of the process is as follows (and shown in Figure 2). Prior to starting the process, the source code, and

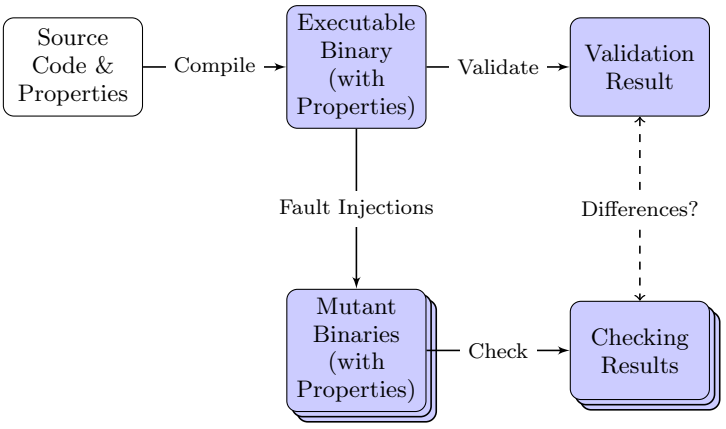

Figure 2: Process Diagram

the properties represented by annotations within the source code, must be defined. The preparation step for the process is then to compile the source code and properties to produce an executable binary in a manner that preserves the properties as annotations. The properties are validated to hold on the executable binary using model checking. The executable binary is then injected with a simulated fault to produce a mutant binary. The properties are then checked upon the mutant binary again using model checking. A difference in the results of validation and checking the properties indicates a vulnerability to the simulated fault injection. The rest of this section details this process.

The choice to start the preparation with the source code and not the binary is made for illustrative clarity and ease of use for the software developer, since defining properties over binaries is more arduous. However, most aspects of the process do not rely upon this choice, and future work is to be able to start directly from the binary (see Section 8.2)

Since this paper considers fault injection attacks upon the binary it is necessary to compile the source code (and here properties) into an executable binary. This executable binary represents the software application that would be executed by the system in practice. Thus to simulate fault injection attacks on the actual system, the executable binary must be used in the simulation. For the process here the compilation must maintain the properties, and so compilation must maintain the properties as annotations in the executable binary.

The properties are validated to hold upon the executable binary using model checking. This is done to ensure that the executable binary does indeed meet the specification of the properties. If there is some other error in the source code or compilation, this can be detected here and not be (incorrectly) attributed to fault injection vulnerability.

Next is the simulation of the fault injection on the executable binary to produce a mutant binary. This simulates the actual fault injection attack and produces a mutant executable binary that represents the executable binary after the attack has been effected. For simplicity here, the fault injection is chosen to always avoid the annotations of the properties, this is discussed later in Section 8.2.

Lastly, the validation results from model checking the executable binary are compared with the checking results from

model checking the mutant binary. Differences here indicate that the fault that was injected yields a change in behaviour that violates the properties, and so could be exploited by an attacker.

Observe that this process describes a formal approach to test for a single simulated injected fault. The process can be repeated many times over with different simulated injected faults to obtain as exhaustive coverage of fault injection attacks as desired (limited only by computation resources, see Section 6.2). Thus, the process can be used for testing complete coverage by all attacks, or focused upon the attacks that are most feasible, common, etc. as suits the goals of fault injection vulnerability detection (for some discussion of this see Section 8.2).

5. IMPLEMENTATION

This section presents the implementation of the process from the previous section. The implementation here exploits currently available tools where possible, despite some having significant limitations. This choice was made in order to focus upon a simple and feasible implementation of the process. For discussion of the limitations of the tools used in the implementation here, please refer to Section 8.2.

The presentation here is highly detailed to allow easy reproduction. This choice was made to maximise clarity and opportunity for others to implement the process for themselves and conduct their own experiments. The rest of the section explicitly details how to implement the process with the chosen tools.

An overview of the implementation is as follows, and shown in Figure 3. The implementation begins with the source code written in the C language and the properties represented in the source code by assert statements. The source code and properties are compiled to an executable binary by the GNU C Compiler (GCC) [14]. The executable binary (including the properties contained within) is transformed into an *intermediate representation* in Low Level Virtual Machine Intermediate Language (LLVM-IR) by the Machine Code Semantics (MC-Sema) tool [44]. The properties are then checked on the intermediate representation using the Low Level Bounded Model Checker (LLBMC) [26, 40]. The executable binary is manually edited to produce the mutant binary file according to the fault model chosen. The steps to model check the properties on the executable binary are then repeated for the mutant binary. Finally, the results of model checking the executable binary and the mutant binary are compared for differences. The rest of this section details this implementation.

5.1 Source Code & Properties

The implementation starts with the source code written in the C language, including the properties to be validated and checked that are expressed as assert statements in this source code. For example, the source code of the motivating example (see Section 2) could have the following property (recalled from Section 3.3)

```
__llbmc_assert(i == 4);
```

that the loop counter `i` reaches `4` inserted between lines 9 and 10. This would check that `i` reaches the value `4` before doing the conditional to test whether access should be granted on lines $10 - 12$.

5.2 Compilation

The compilation from source code to executable binary for this paper is done with GCC. Note that here a listings file is also generated with annotations that will be exploited to do the fault injection later. The following is the command used to compile with GCC for this paper.

```
$gcc -m32 -ggdb -c -Wa,-a,-ad -o test.o
     test.c > test.lst
```

Here `-m32` specifies compiling for 32-bit architecture. The `-ggdb` argument includes debugging information that will be used to help translate the intermediate language later in the implementation. The `-c` argument indicates to compile and assemble the source code, but do not link (this simplifies the scope of checking since no library code is linked at this stage). The `-Wa,-a,-ad` argument specifies annotations to output that will be used later to do the fault injection (`-a` to turn the listing on, `-ad` to omit unnecessary debug information). The `-o` is used to specify the output file (`text.o`) for the executable binary. Here `test.c` is the source file with properties. Lastly, `> test.lst` outputs the annotations used later to do the fault injection into the file `test.lst`.

Note that the above command preserves the assert statements along with the debugging information, so these can be exploited in later stages.

5.3 Intermediate Representation

The translation from executable binary to LLVM-IR is done by MC-Sema in two stages. The first stage uses the executable binary to generate a Control Flow Graph (CFG). The second stage uses the CFG to generate the LLVM-IR.

Executable Binary to CFG

The first stage is done by the `bin_descend` tool (included within MC-Sema) using the below command.

```
$bin_descend -march=x86 -d
    -func-map="test_map.txt"
    -entry-symbol=checkPIN -i=test.o
```

Here the `-march=x86` argument specifies X86 architecture. The `-d` flag enables output of debugging information, used in later stages of the implementation. The `-func-map= "test_map.txt"` argument informs of the file (`test_map.txt`) that contains specifications of externally referenced functions (e.g. `__llbmc_assert 1 C N` to indicate that the function `__llbmc_assert` has 1 argument, `C` to represent the calling convention here is for CleanUp to clean up the stack after the function call, the `N` to mention that the function has a return). The `-entry-symbol=checkPIN` argument indicates the function name of the entry point into the code, here the `checkPIN` function. Lastly, `-i=test.o` indicates to input from the file `test.o`.

CFG to LLVM-IR

The second stage is to translate the CFG to LLVM-IR. This is done by the `cfg_to_llvm` tool also included in MC-Sema. The command to achieve this is shown below.

```
$cfg_to_bc -mtriple=i686-pc-linux-gnu
    -driver=test_entry,checkPIN,0,return,C
    -o test.bc -i test.cfg
```

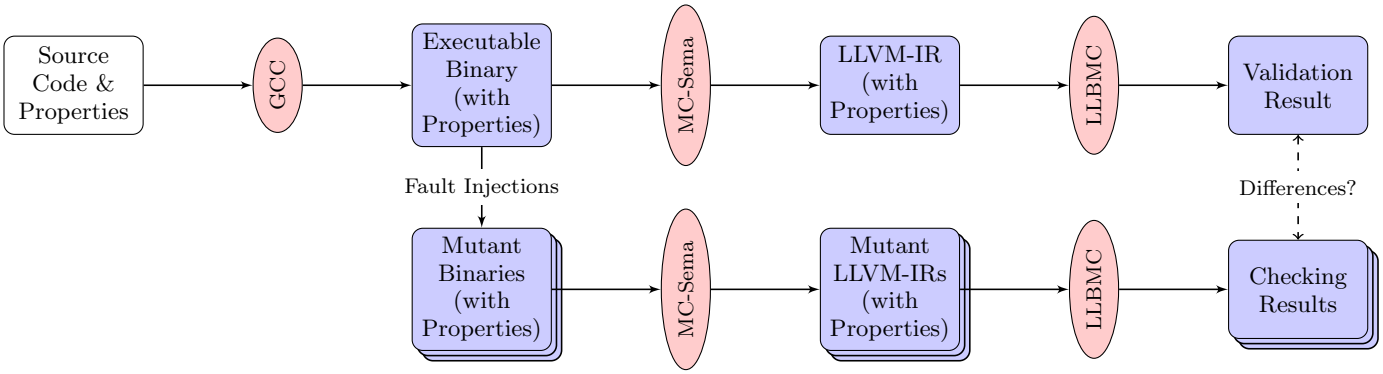

Figure 3: Implementation Diagram

Again `-mtriple=i686-pc-linux-gnu` indicates the X86 architecture (on linux). The argument `-driver=test_entry, checkPIN,0,return,C` defines the entry point in the generated LLVM-IR to be `test_entry` and this should correspond to the entry point symbol `checkPIN` in the CFG, the `0` represent the argument count, the `return` to specify that the function has a return, and finally the `C` represents the calling convention. As usual `-o` indicates the output file name (here `test.bc`). Lastly, `-i` is the input file name (here `test.cfg`).

5.4 Model Checking

The model checking of properties on the intermediate representation is done by LLBMC. The command used here to model check with LLBMC is as below.

```
$llbmc -function-name=test_entry
    --ignore-missing-function-bodies
     --max-loop-iterations=20
     -only-custom-assertions test.bc
```

The argument `-function-name=test_entry` makes certain that LLBMC checks the specified function (and not others). The `-ignore-missing-function-bodies` argument is used to ignore missing functions, such as those that were not linked and so do not appear in the LLVM-IR. The argument `-max-loop-iterations=20` specifies bounds for the model checking, here limited to 20 loop iterations. The `-only-custom-assertions` argument forces LLBMC to only check the properties specified, and not other default properties. Lastly, `test.bc` is the input file.

Note that the above steps from executable binary to model checking can be repeated (with changed file names) for the mutant binary, and so will not be repeated below.

5.5 Fault Injection

The fault injection attack that takes an executable binary and yields a mutant binary is done by manually editing the file. To achieve this the listing file generated by GCC during compilation is exploited to find which locations in the executable binary correspond to the source code, and thus easily understand the semantics so as to be able to simulate specific fault injection attacks. Once the locations can be found with the listing, the binary can be modified using any binary editing tool and the modified one saved as the mutant binary.

```
Result:
=======
No error detected.
```

Figure 4: Property 1 : executable binary verification result

```
Result:
=======
Error detected.

Error synopsis:
===============
Assertion failed: Custom assertion (assert
     or __llbmc_assert) does not hold.

Error location:
===============
Error occurs in basic block "block_0x10b"
    of function "sub_0".
No debug information available.

Stack trace:
============
#0 void @sub_0(%struct.regs* %0)
#1 i32 @demo_entry()
```

Figure 5: Property 1 : mutant binary verification result

5.6 Detecting Vulnerability

Once the results of model checking have been produced for both the executable binary and the mutant binary, fault injection vulnerabilities can be detected when these results differ. In Figure 4 the output of LLBMC is shown for when all the properties hold. By contrast Figure 5 shows the LLBMC output when a property is violated. Note that due to compilation to binary and then translation to LLVM-IR, LLBMC is unable to gather sufficient information to produce a useful trace of the property violation.

6. EXPERIMENTAL RESULTS

This section presents experimental results obtained by ap-

plying the process using the implementation. For clarity, all the attacks and properties presented here use the motivating example (of Section 2). The experimental results show that by using the process presented here (and the implementation detailed in Section 5) a variety of fault injection vulnerabilities can be detected. The rest of this section presents illustrative 1-bit fault injection attack examples building to the general solution that can detect many kinds of fault injection attacks, and concludes with a variety of different kinds of fault injection attacks to illustrate generality of the process and implementation.

Attack 1

The first attack to consider is the attack detailed in Section 2 where changing a single bit changes the `PINSize` variable from `4` to `0`. Recall that since the loop that checks the digits of the PINs iterates from `i = 0` to `i < PINSize`, that the loop will be skipped (since `0 < 0` does not hold). This fault injection attack can be exploited by the attacker since the PINs are never checked against each other. Therefore, any candidate PIN will lead to access being granted, and so the attacker can use this fault injection attack to gain access (even when the candidate PIN they supply does not match the true PIN).

Property A

A simple property to detect such an attack would be to ensure that `i` reaches `4`. This can be achieved by taking the following property (recalled from Section 3.3):

```
__llbmc_assert(i == 4);
```

inserted between lines 9 and 10.

With this property added to the source code the process was repeated with attack 1. The results of model checking the mutant binary reveal that the assertion was violated and thus the model checking result different form the validation result. Thus, vulnerability to the first fault injection attack was detected.

Attack 2

An alternative 1-bit fault injection attack that has the same effect as Attack 1 is to initialise the value of the variable `i` to `4`. This will again grant access even when the two PINs differ since the loop will be bypassed as before (this time since `4 < 4` does not hold). Observe that since `i` is intiailised to `4` this fault injection attack should not violate the assert statement of Property A. This can be exploited by the attacker in the same manner as Attack 1 to gain access with a candidate PIN that does not match the true PIN.

The process was repeated with Attack 2 and Property A. As expected, the fault injection vulnerability was not detected.

Property B

The above result illustrates that the choice of properties need to consider the behaviour of the program rather than focus on particular variables that are incidental to the program's execution. Thus, a property that captures the idea that unequal PINs should never lead to access being granted could be defined as follows (also recalled from Section 3.3).

```
__llbmc_assert(    !(PINCandidate != PINTrue)
               || grantAccess == false);
```

where this property is inserted into the motivating example code after line 12. This property expresses that if the two PINs are different then the access is not granted.

The process was then repeated for both Attack 1 & 2, and with Property B. As expected Property B was able to detect both fault injection vulnerabilities represented by Attacks 1 & 2. This shows that considering the behaviour is more important than considering the variables used to achieve the behaviour. That is, properties should consider `PINCandidate`, `PINTrue`, and `grantAcces` rather than `i` or `badValue`.

Attack 3

Observe that Property B detects attacks that allow access when the PINs are not equal, but does not consider when the PINs are equal. An alternative attack could be to deny access even to a user who knows the correct PIN. Consider the 1-bit fault injection attack that changes the value of `true` (represented in binary by `0...01`) to `false` (represented in binary by `0...00`).

Now consider this attack upon the motivating example line 11, changing `grantAcces = true` to be

```
grantAccess = false;
```

and thus preventing any access even when the PINs are equal.

The process was then repeated using this new Attack 3 with Properties A & B. As expected neither property was able to detected this attack. Property A failed since the attack does not effect the iterator `i`. Property B failed since when the PINs are equal no further behaviour is considered.

Property C

To also account for Attack 3, the original Property B needs to be extended to also consider the behaviour when the PINs are equal. Indeed, the ideal behaviour of the code can be represented by the following property.

```
__llbmc_assert(   (   !(PINCandidate != PINTrue)
                   || grantAccess == false)
               && (   !(PINCandidate == PINTrue)
                   || grantAccess == true));
```

This ensures that when the PINs are unequal access is not granted, and when the PINs are equal then access is granted. Property C is added to the motivating example in the same place as Property B would be; after line 12.

The process was then repeated with all three Attacks (1, 2 & 3) using Property C. As expected Property C was able to detect all three fault injection attacks.

6.1 Other Attacks

This section considers several more fault injection attacks in less detail than those above. These include several more 1-bit fault injection attacks, and then other kinds of attacks, culminating in an attack that can only be effected in the binary and not in the source or by "compiling" directly to an intermediate representation.

There are several other 1-bit fault injection attacks that can be performed against the motivating example. Such attacks include changing the initialisation value of variables such as `grantAccess` and `badValue`, e.g. at line 6 changing the initialisation `badValue = true` to instead be `badValue`

= false. These are all detected by at least Property C, if not also Property B.

A different kind of attack that targets the control flow of the program is to change the target of a jump instruction. For example, the jump from the conditional on line 5 of the motivating example, could change from skipping the following instruction on line 6, to always executing line 6. Thus badValue = true (line 6) would always be executed, regardless of the outcome of the conditional. This can be done by modifying three bits (or one byte) of the target (relative) address of the jump, from 0000 0111 to 0000 0000. This attack was successfully detected by Property C (but not Properties A or B).

A more significant attack on the behaviour of an instruction is to simply change the instruction to a NOP (non-operation). Consider in the motivating example when a fault injection changes the instruction that represents line 6 badValue = true; to a NOP. This requires modifying 4 bytes (on the X86 32-bit architecture used here). The change would allow access for any candidate PIN (by never recording differences to badValue). This attack was successfully detected by Properties B and C (but not Property A).

Instead of changing a subtle behaviour like a jump, or simply wiping an instruction to a NOP, another fault injection attack is to change the instruction type, e.g. changing a CMP instruction to a MOV instruction. This can be done on the CMP instruction that compares the values PINCandidate[i] != PINTrue[i] on line 5 of the motivating example. This requires modifying 3-bits of the executable binary, from 0011 1011 to 1000 1011. The result of this change is that the following line that sets badValue to true will always be executed. This change prevents access even when the correct candidate PIN is provided, similar to Attack 3. As expected, this attack is successfully detected by Property C (but not Properties A or B).

One attack that is of particular interest here, is an attack that can be represented in the executable binary but not in source code or from "compiling" the source code to an intermediate language, such as C and LLVM-IR, respectively. An example of this kind of attack is the modification of the return value stored in the return register (eax on X86 architectures). This kind of attack can be simulated and detected using the process and implementation here.

To properly handle this attack requires a function call, and so the motivating example is modified by placing the code in Section 2 inside a function that returns grantAccess. The attack works by altering the returned value from this new function. The return value is stored in the eax register, for the motivating example this is handled by the binary operation corresponding to the assembly instruction below.

```
mov    eax, DWORD PTR [ebp-0x8]
```

The attack is then to change the value loaded into eax so that the function behaviour is changed. For example, by changing the value 0000 1000 to 0000 1100 the returned value of grantAccess can change from False to True, so the access will be granted even if the two PINs are not equal. (Note that if the returned value is already 0000 1100 then this can be ignored, or the value changed to 0000 1000 inverting the function behaviour). This attack is detected by Property C, although Property C needs to be located outside the function call so as to check the value of grantAccess after the function return (or more precisely the returned value that corresponds to grantAccess).

These attacks illustrate that the process and implementation here are not limited to a single fault model or kind of fault injection attack. Thus, the same process and implementation can be used for a variety of fault models and fault injection attacks, as long as some consideration is taken to choose the right properties. Further, the process and implementation here can detect fault injection attacks that cannot be found by checking the source code or intermediate representation alone; the executable binary must be part of the process and implementation.

6.2 Computational Data

This section discusses the experimental results of automating the the process and implementation of this paper. This automation is straightforward once all the tools are available.

To support the automation experiments, a simple fault injection tool was created[1]. This tool takes an executable binary, and produces mutant binaries by replacing one byte with 0. This fault injection model is naive, but simple to implement and conduct experiments with to test the automation of the process and implementation.

To estimate the feasibility of finding arbitrary fault injection vulnerabilities in a executable, the following experiment was conducted. A script was written[2] that takes a source code and properties written in C, and compiles these to an executable binary with GCC. This executable binary is then validated to ensure the properties hold. The fault injection tool was used to generate mutant binaries for each possible byte in the executable binary being set to 0. The script then enters into a loop over the mutant binaries that generates the LLVM-IR for the mutant binary and model checks the properties on this LLVM-IR. The result of this model checking is then used to determine if the injected fault found a fault injection vulnerability. The runtime and number of fault injection vulnerabilities was counted and reported at the end of the experiment.

This script was run over a version of the motivating example (from Section 2). The fault injection tool was limited to injecting faults into the .text area of the executable binary that corresponds to the compiled source code (to reduce time wasted model checking fault injection vulnerabilities in the header or other unrelated parts of the executable binary).

The executable binary produced by GCC in this case was 3024 bytes. The .text area was 159 bytes and thus 159 1-byte fault injection attacks were simulated, yielding 159 mutant binaries. The following experiments were conducted on a virtual machine configured with one CPU, and 7662 MB of RAM running Linux Ubuntu 14.04 LTS. The virtual machine was hosted on a Macbook Pro with 3,1 GHz Intel Core i7 processor, 16 GB of RAM, running OS X EL Capitan 10.11.

Three different experiments were conducted, testing the three different properties presented earlier. The first experiment with Property A detected 36 fault injection vulnerabilities, and had a runtime of 7 minutes. The second experiment with Property B detected 37 fault injection vulnerabilities, and had runtime of approximately 1 hour. The third experiment with Property C detected 37 fault injection

[1]The code for this tool is available upon request.
[2]This script is available upon request.

vulnerabilities, and had a runtime of approximately 2 hours. The main cost in time was the model checking by LLBMC, as is clearly shown by the significant difference made by the choice of property.

Observe that the number of fault inject attacks to test in this manner is linear in the size of the binary executable. Thus, automatically testing all such fault injection attacks is feasible, particularly since the implementation can be easily run in parallel.

7. RELATED WORK

This section discusses some related works and their differences with respect to the process presented here.

One of the first uses of formal methods to analyse fault injection vulnerabilities was to verify a counter measure in the implementation of CRT-RSA by analysing the C source code [11]. The authors show that by adding ANSI/ISO C Specification Language (ACSL) [7] properties to the CRT-RSA pseudocode, they could verify that the Vigilant's CRT-RSA countermeasure sufficiently protects against fault injection attacks. Although one of the first to use formal methods for detection and protection against fault injection attacks, the approach is quite limited. The lack of fault injection upon the executable binary limits to attacks that can be considered on the source code alone. Thus, the last attack presented in Section 6.1 would not be detectable by this approach. Further, the analysis was only for a single countermeasure to prove it worked, rather than to consider all possible fault injection attacks and models as is the goal of the process presented here.

Perhaps the closest to the process presented here is the Symbolic Program Level Fault Injection and Error Detection Framework (SymPLFIED) [31] a program-level framework to identify potential vulnerabilities in a software. The vulnerabilities are detected by combining symbolic execution and model checking techniques. However, the SymPLFIED framework is limited SymPLFIED only supports the MIPS architecture [33]. One of the proposed future work in [47] was building front-end to support X86 architecture, but to the best of our knowledge, no further work has been published on supporting new architectures in SymPLFIED.

In [32], Potet et al. present Lazart, a tool that can simulate a variety of fault injection attacks and detect vulnerabilities using formal methods. The Lazart process begins with the source code which is compiled to LLVM-IR. The simulated fault is created by modifying the control flow of the LLVM-IR. Symbolic execution is then used to detect differences in the control flow, and thus detect vulnerabilities. Although this high level approach is similar to that of this paper, there are several limitations with respect to the process presented here. Again the lack of fault injection upon the executable binary limits to Lazart to attacks that can be considered on the LLVM-IR and control flow, and so the last attack presented in Section 6.1 would not be detectable by Lazart. Further, the choice of symbolic execution does not account for concrete values, and thus is less complete than the model checking used here [30, 42].

In [37] the authors propose combining the Lazart process with the Embedded Fault Simulator (EFS) [8]. This extends from the capabilities of Lazart alone by adding lower level fault injection analysis that is also embedded in the chip with the program. The simulation of the fault is performed in the hardware, so the semantics of the executed program correspond to the real execution of the program. However, EFS is limited to only considering fault injection attacks that implement a single fault model - the instruction skip fault model.

An entirely low level approach is taken by Moro et al. [29] who use model checking to formally prove the correctness of their proposed software countermeasures schemes against fault injection attacks. The approach has some similarities to that of this paper, in using model checking while focusing on low level representations. However, the focus is on a very specific and limited fault injection model that causes instruction skips and ignores other kinds of attacks. Further, the model checking is over only limited fragments of the assembly code, and not the program as a whole.

A less formal approach is taken in [1] where experiments are used for testing the TTP/C protocol in the presence of faults. Rather than attempting to find fault injection attacks, they injected faults to test robustness of the protocol and simulation of hardware faults. They combined both hardware testing and software simulation testing, comparing the results as validation of their approach.

<<<<<<< HEAD A more fault model inference focused approach is taken by Dureuil et al. [13]. They fix a hardware model and then test various fault injection attacks based upon this hardware model. Fault detection is limited to EEPROM faults on the ARMv7-M architecture. The fault model is then inferred from the parameters of the attack and the embedded program. The faults are simulated upon the assembly code and the results checked with predefined oracles on the embedded program. Although the approach uses neither formal methods nor general fault models, the choice of fault model and derivation of this provides some interesting guidance for selection fault models and our future work.
======= A more fault model inference focused approach is taken by Dureuil et al. [13]. They fix a hardware model and then test various fault injection attacks based upon this hardware model. Fault detection is limited to EEPROM faults on the ARMv7-M architecture. The fault model is then inferred from the parameters of the attack and the embedded program. The faults are simulated upon the assembly code and the results checked with predefined oracles on the embedded program. Although the approach uses neither formal methods nor general fault models, the choice of fault model and derivation of this provides some interesting guidance for selection fault models and our future work. >>>>>>> b06bc9c948806aec9d82110b9a2b94e27f623565

8. CONCLUSIONS

Fault injection has recently been increasingly used to attack software applications, and test system robustness. This paper presents a formal process that uses model checking to detect fault injection vulnerabilities in binaries. This process supports the detection of many varieties of fault injection vulnerabilities, and does not rely on any particular system architecture, fault model, or other restricted choices (as are common in the literature).

A detailed implementation of this process demonstrates the feasibility, versatility, and strength of the process. The implementation shows that the process can be implemented with existing tools in a straightforward and reproduceable manner.

Experimental results show that a variety of fault injection vulnerabilities can be detected using the process and imple-

mentation. These are detailed on a small illustrative example to clearly illustrate the process and the results. These also include an attack upon the executable binary that does not appear in the source code (or "compilation" of the source code into an intermediate representation such as LLVM-IR). Further, the experimental results show that once an appropriate property is chosen, detecting all the attacks here can be done in a uniform manner. Lastly, the experimental results show that automating the process is feasible, with reasonable runtimes even for consumer hardware and without any optimisation.

The rest of this section discusses the choices made in this paper and future work.

8.1 Discussion

This section discusses the choices made in designing the process and with the implementation. Many of these are influenced by the larger goal of automating the detection of fault injection vulnerabilities in executable binaries.

The preparation point for the process *here* is the source code. This choice was made for two main reasons. The first reason was for simplicity in specifying the properties and understanding the behaviour of the binary. While properties can be specified on the executable binary directly, this is more arduous for the specifier and more distant from the semantics of the source code. The second reason is that there are very few model checkers that can operate directly on executable binaries [25, 39]. Although model checking of some executable binaries is possible, the limitations imposed were too strict for the general approach considered here. In particular, the model checkers that can operate directly upon binaries have limitations in their ability to handle larger binary files representing more complex programs, and do not always have the capability to handle all the properties to to be checked [25, 39, 9]. For these reasons future work is to start the process with the executable binary.

The choice to add the properties to the source code was made here for simplicity (and for ease of property specification as above). This is not a good choice in general for several reasons. One reason is that fault injections attacks could modify the properties (if embedded in the binary) interfering with the process since this may indicate a vulnerability when only the property has changed. Another reason is that the properties must be maintained throughout the compilation and then maintained again from the executable binary to the intermediate language. Yet another reason is that adding the properties to the executable binary may change the structure in such a way that either introduces or prevents fault injection attacks that would occur on the executable binary without the properties. For these reasons future work will adapt the process to accept the properties as a separate input to the model checking stage of the process, and not have them embedded into the source code or executable binary.

The choice here to use MC-Sema was to be able to work on the LLVM-IR. The choice of LLVM-IR is due to this being a widely used intermediate representation that is supported by many other tools. However, there are limitations with MC-Sema that may limit future work with the implementation used in this paper. MC-Sema supports only the X86 architecture, and does not even support all X86 instructions [44]. The use of X86 architecture throughout this paper is driven by the use of MC-Sema, future work is to be able to implement the process for other architectures, particularly ARM since ARM is widely used on embedded devices [18].

The LLBMC model checker is sufficient for the safety properties (represented by assert statements), but does not support liveness properties. Thus although LLBMC was sufficient for the proof of concept here, future work will exploit a non-bounded model checker that can also accept liveness properties.

Another limitation of the combination of MC-Sema and LLBMC here is the inability to produce traces that illustrate how a property has been violated. This is not significant in a highly automated environment where merely finding the vulnerability is the goal. However, when modifying the source code to avoid the vulnerability, or when finding the exact weakness in more complex programs, then the trace can be extremely helpful. Therefore, traces would be desirable to have as a possible output in future.

The fault injections here were done manually. However, the larger goal envisioned in this work is to automate the detection of fault injection vulnerabilities. Therefore, a fault injection tool that can be automated will be required for this automated detection. There exist some tools that claim to do fault injection simulation [17, 23], however most of these are constrained by limited fault models, and further, the majority do not operate on the executable binary representation of the program.

8.2 Future Work

Future work is to develop a fault injection tool that can easily fit into an automated tool chain implementing the process presented here. This would resolve the issues with the non-automated approach as mentioned above, and also stimulate other areas of research (see below).

There already exist several tools to simulate fault injection attacks on software [17, 19]. However, these tools are limited by various choices that make unsuitable for the process here (hence their lack of use in the implementation). Several are only able to inject faults into intermediate representations, and not into executable binaries [43, 32, 46], thus being unable to simulate faults that appear only at the executable binary level. Others have different limitations, such as: specific hardware platforms [31, 36], specific source code languages [24, 11, 46], or requiring simulating drivers [12]. Despite these limitations, many include useful techniques or developments that could be incorporated into a general fault injection tool for executable binaries.

A complementary area of research is to explore where to perform fault injections that intelligently modify the behaviour. The main challenge with fault injection attacks is to find the appropriate hardware effect to yield a viable and useful software effect [35]. It is easy to inject a fault that changes the behaviour of the software, but more complex to inject a fault that changes the behaviour in a manner beneficial to an attacker. Also for automating the discovery of fault injection vulnerabilities, computational costs need to be considered. Merely injecting every possible 1-bit fault injection attack on a small executable binary may be feasible, but when multiple fault injection models are considered, or multiple fault injection attacks on the same executable binary, then this rapidly becomes infeasible. Thus, developing further techniques to discover the most significant fault injection attacks, or exploiting the properties to consider the most significant components of the executable binary should

improve the effectiveness of an automated process.

9. REFERENCES

[1] A. Ademaj, P. Grillinger, P. Herout, and J. Hlavicka. Fault tolerance evaluation using two software based fault injection methods. In *On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International*, pages 21–25. IEEE, 2002.

[2] P. Andouard. *Outils d'aide à la recherche de vulnérabilités dans l'implantation d'applications embarquées sur carte à puce*. PhD thesis, Bordeaux 1, 2009.

[3] C. Baier, J.-P. Katoen, and K. G. Larsen. *Principles of model checking*. MIT press, 2008.

[4] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *IACR Cryptology ePrint Archive*, 2004:100, 2004.

[5] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[6] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. DiVinE 3.0–an explicit-state model checker for multithreaded C & C++ programs. In *International Conference on Computer Aided Verification*, pages 863–868. Springer, 2013.

[7] P. Baudin, J.-C. Filliatre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C specification language, version 1.4. *CEA*, 6, 2009.

[8] M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant. Idea: embedded fault injection simulator on smartcard. In *International Symposium on Engineering Secure Software and Systems*, pages 222–229. Springer, 2014.

[9] S. Biallas, J. Brauer, and S. Kowalewski. Arcade.PLC: A verification platform for programmable logic controllers. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 338–341. ACM, 2012.

[10] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[11] M. Christofi, B. Chetali, and L. Goubin. Formal verification of an implementation of CRT-RSA vigilant's algorithm. In *PROOFS Workshop: Pre-proceedings*, page 28, 2013.

[12] K. Cong, L. Lei, Z. Yang, and F. Xie. Automatic fault injection for driver robustness testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 361–372. ACM, 2015.

[13] L. Dureuil, M.-L. Potet, P. de Choudens, C. Dumas, and J. Clédière. From code review to fault injection attacks: Filling the gap using fault model inference. In *International Conference on Smart Card Research and Advanced Applications*, pages 107–124. Springer, 2015.

[14] B. Gough. *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[15] S. Guilley, L. Sauvage, J.-L. Danger, and N. Selmane. Fault injection resilience. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pages 51–65. IEEE, 2010.

[16] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner. QEMU-based fault injection for a system-level analysis of software countermeasures against fault attacks. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 530–533. IEEE, 2015.

[17] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, Apr. 1997.

[18] D. Jaggar et al. ARM architecture and systems. *IEEE micro*, 17(4):9–11, 1997.

[19] A. Johansson. Software implemented fault injection used for software evaluation. *Building Reliable Component-Based Systems*, 2002.

[20] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Proactive detection of computer worms using model checking. *IEEE Transactions on Dependable and Secure Computing*, 7(4):424–438, 2010.

[21] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification*, pages 423–427. Springer, 2008.

[22] J. Kinder and H. Veith. Precise static analysis of untrusted driver binaries. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 43–50. FMCAD Inc, 2010.

[23] M. Kooli and G. Di Natale. A survey on simulation-based fault injection tools for complex systems. In *Design & Technology of Integrated Systems In Nanoscale Era (DTIS), 2014 9th IEEE International Conference On*, pages 1–6. IEEE, 2014.

[24] J.-B. Machemie, C. Mazin, J.-L. Lanet, and J. Cartigny. SmartCM a smart card fault injection simulator. In *2011 IEEE International Workshop on Information Forensics and Security*, pages 1–6. IEEE, 2011.

[25] T. Mehler and P. Leven. Introduction to StEAM-an assembly-level software model checker. Technical report, Technical Report 193, University of Dortmund and University of Freiburg, 2003.

[26] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'12, pages 146–161, Berlin, Heidelberg, 2012. Springer-Verlag.

[27] N. Moro. *Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2014.

[28] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 77–88. IEEE, 2013.

[29] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, 2014.

[30] C. S. Păsăreanu and W. Visser. *Symbolic Execution*

and Model Checking for Testing, pages 17–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[31] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer. SymPLFIED: Symbolic program-level fault injection and error detection framework. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 472–481. IEEE, 2008.

[32] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil. Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 213–222. IEEE, 2014.

[33] C. Price. MIPS iv instruction set, 1995.

[34] T. Reinbacher, M. Kramer, M. Horauer, and B. Schlich. Challenges in embedded model checking-a simulator for the [mc] square model checker. In *2008 International Symposium on Industrial Embedded Systems*, pages 245–248. IEEE, 2008.

[35] L. Rivière. *Securing software implementations against fault injection attacks on embedded systems*. PhD thesis, TELECOM ParisTech, Paris, september 2015.

[36] L. Riviere, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, and L. Sauvage. High precision fault injections on the instruction cache of ARMv7-M architectures. In *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*, pages 62–67. IEEE, 2015.

[37] L. Rivière, M.-L. Potet, T.-H. Le, J. Bringer, H. Chabanne, and M. Puys. Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks. In *International Symposium on Foundations and Practice of Security*, pages 92–111. Springer, 2014.

[38] K. Rothbart, U. Neffe, C. Steger, R. Weiss, E. Rieger, and A. Mühlberger. High level fault injection for attack simulation in smart cards. In *Test Symposium, 2004. 13th Asian*, pages 118–121. IEEE, 2004.

[39] B. Schlich and S. Kowalewski. [mc] square: A model checker for microcontroller code. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 466–473. IEEE, 2006.

[40] C. Sinz, F. Merz, and S. Falke. LLBMC: A bounded model checker for LLVM's intermediate representation - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 542–544, 2012.

[41] V. Štill, P. Ročkai, and J. Barnat. DIVINE: Explicit-state LTL model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 920–922. Springer, 2016.

[42] T. Su, Z. Fu, G. Pu, J. He, and Z. Su. Combining symbolic execution and model checking for data flow testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 654–665. IEEE, 2015.

[43] A. Thomas and K. Pattabiraman. LLFI: An intermediate code level fault injector for soft computing applications. In *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

[44] Trail of bits. Mc-semantics, 2016. https://github.com/trailofbits/mcsema.

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

[46] S. K. Vishal Chandra Sharma, Ganesh Gopalakrishnan. Towards reseiliency evaluation of vector programs. In *21st IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*, 2016.

[47] F. Q.-Y. Yuan. *Formal framework and tools to derive efficient application-level detectors against memory corruption attacks*. PhD thesis, University of Illinois at Urbana-Champaign, 2010.