UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

Systematics validation of a software hardening framework

CONFIDENTIAL

This thesis and its annexes, including the source code of the developed software, contain confidential information. It cannot be distributed. This restriction expires June 30, 2020.

Supervisor: Charles Pecheur
Readers: Gautier Dallons
Marc Lobelle

Thesis submitted for the Master's degree
in computer science (120 credits)
options: software engineering and programming systems
by Alex Vermeylen

Louvain-la-Neuve
Academic year 2014-2015

Contents

1 Introduction **1**

2 Software hardening **3**
2.1 Hardening techniques overview . . . 3
2.2 Hardening rules . . . 4
2.2.1 Example program . . . 4
2.2.2 Data hardening . . . 5
2.2.3 Control flow hardening . . . 6
2.2.4 Hardened example . . . 8
2.3 Tests . . . 9

3 Fault injection **13**
3.1 At hardware level . . . 13
3.1.1 Injection with contact . . . 13
3.1.2 Injection without contact . . . 13
3.2 At software level . . . 14
3.2.1 Software fault injection . . . 14
3.2.2 Simulation . . . 15
3.3 Comparison criteria . . . 16
3.3.1 Fault representativeness . . . 16
3.3.2 Comparison criteria . . . 17
3.3.3 Comparison . . . 18

4 Mutation testing **21**
4.1 The concept . . . 21
4.1.1 Mutation operators . . . 21
4.2 Weaknesses . . . 22
4.2.1 Efficiency . . . 23
4.2.2 Computation of the mutation score . . . 24
4.3 Mutation technique and fault injection . . . 24
4.3.1 Evaluation . . . 24

5 Related work **27**
5.1 The TIMA Lab. hardener prototype . . . 27
5.1.1 Context . . . 27
5.1.2 The test approach . . . 27
5.2 The MAintainable Real-Time System (MARS) . . . 29
5.2.1 The project . . . 29
5.2.2 Evaluation of error detection mechanisms . . . 29
5.3 Gpu-qin . . . 30
5.3.1 Fault injection methodology . . . 30
5.4 Existing fault injection tools . . . 31
5.4.1 Xception . . . 31
5.4.2 DOCTOR . . . 31
5.4.3 FTAPE . . . 32
5.5 Comparison . . . 32

6 Problem statement **35**
6.1 Context . . . 35
6.1.1 Errors . . . 35
6.1.2 Constraints . . . 36
6.1.3 Test cases . . . 36
6.2 Main approach . . . 37
6.2.1 Choice . . . 37
6.2.2 Concept . . . 38

7 Proposed solution **41**
7.1 Methodology . . . 41
7.2 Functional architecture . . . 41
7.3 Technical architecture . . . 42
7.4 Framework . . . 43
7.4.1 Data flow . . . 43
7.4.2 Control flow . . . 46
7.5 Strategies . . . 48
7.5.1 Define a strategy . . . 48
7.5.2 parameters . . . 49
7.6 Generation . . . 49
7.7 Cunit . . . 49
7.8 Tests scripts . . . 50
7.8.1 Organization . . . 51
7.8.2 Run the tests . . . 51
7.8.3 Results . . . 51
7.9 Conclusion . . . 52

8 Validation **53**
8.1 Benchmarks . . . 53
8.1.1 Measures . . . 53
8.1.2 Analyze . . . 54
8.2 Experiments . . . 55
8.2.1 Fault injections . . . 55
8.2.2 Strategies . . . 58
8.3 Limitations . . . 60
8.4 Conclusion . . . 61

9 Future work **63**
9.1 False positive result . . . 63
9.2 Execution path . . . 63
9.3 Results . . . 64
9.4 Running options . . . 64
9.4.1 Time-out . . . 64
9.5 Threads . . . 64
9.6 Unit testing frameworks . . . 64
9.7 Errors . . . 65
9.7.1 Error models . . . 65
9.7.2 Combining errors . . . 65
9.8 Combining tests approaches . . . 65
9.9 Inter-functions injections . . . 65
9.10 Conclusion . . . 65

Abstract

This master thesis discusses the application of a fault injection testing methodology to robust programs. The objective was to find and implement a fault injection-based solution that allows to evaluate the robustness of programs under tests against a set of defined error types. This approach also permits to challenge the error detection mechanisms implemented in the program to test. By comparing the strengths and weaknesses of existing approaches and tools, a methodology have been chosen. Based on some representations of the source code, the instructions are analyzed independently to find the relevant spots to inject adequate faults. The implemented methodology have a high controllabillity as it allows precise faults to be injected on both data and control flow. Moreover, the tester is able to define and configure his own test strategies. Finally, the document provides some results of experiments on different kinds of source code. These have shown the genericity of the methodology. Furthermore, they demonstrated the efficacy of the fault injection technique considered as the results proved the robustness of different programs for specific types of errors.

Acknowledgments

I would like to thank my advisor Charles Pecheur for his support and advice throughout the whole academic year.

I also want to especially thank Gautier Dallons, Dimitri Durieux and Christophe Ponsard for their support, assistance and feedback that were strongly helpful.

I am also grateful to Karim Vermeylen and Laura Weerens for reading back this thesis and giving their insightful remarks.

Finally, I would particularly thank the entire Clinch team for the friendly environment they provided in all circumstances.

Chapter 1

Introduction

Nowadays, the use of computers is growing in every area. These processor-based systems are involved in all kinds of environment. Sometimes, an error can occur. It can, for example, comes from the environment (extreme heat, cosmic-ray, etc.), from an electronic dysfunction or a hardware design flaw. These errors will affect the hardware used by the software to store and retrieve data which will lead a program to use corrupted information. Naturally, in the case of a personal computer or a vending machine, the effect will not be serious. However, in the case of safety-critical systems, the consequences can be really dramatic.

An example of such failure in a safety-critical system is the Therac-25. It was a medical equipment used for radiation therapy that gave massive overdose of radiation to some patients, causing the death of six of them [1].

Software hardening represents a way to make a program robust against some kinds of error. The most basic techniques rely on hardware redundancy. Although this approach is very efficient, it is very expensive. Indeed, more hardware pieces need to be bought. Also, sometimes, more expensive specific hardware that supports these kind of error detection mechanisms is required. Moreover, hardware quickly becomes out-of-date and need to be replaced.

Some more advanced approaches make use of software redundancy. These techniques are designed to be used independently of the hardware on which they are executed. The source code is simply modified in such a way that the program implements some error detection mechanisms. Some of these software approaches can be automated. A *hardener* is then used to make robust code from "classical" one. This kind of code will be considered in the context of this thesis.

The faults that can appear during the execution of a program can be of two kinds. The firsts affect the data used by the program to make its computations. The second kind regroup faults affecting the code (Control Flow Errors) causing the program to executes its instructions in a wrong order. Also, one should distinguish between transient and permanent faults. Finally, the physical location of the faults can be taken into account.

Given a robust program, we want to verify that the hardened application can detect the faults against which it is robust. To do so, faults need to be simulated during the execution of the program. After such a run test, the results should show if the error detection mechanisms have been triggered. That allows a tester to draw some conclusions about the robustness of the program against the injected errors. To do so, several techniques exist. However, none of them are perfect. For some, the cost can be very high, for others the coverability is not sufficient.

In this thesis, the different software approaches will be detailed and compared. Based on the possibilities and our needs, one will be chosen and a specific tool implementing it will be developed and tested on different types of robust source code. Two types of hardened program will be used. The first will be source code hardened through the rules described in [2] and detailed in the next chapter. The other one will be provided by the CETIC. As part of a project, this research center have developed a hardener. Some samples of source code hardened through this tool will be used as a concrete test case.

This document is organized as follow. First, some background chapters will allow the reader to fully understand the concepts of software hardening and faults injection. Some fault injection techniques will be detailed. The comparison criteria that can be used to differentiate these approaches will be introduced and the main kinds of faults injection techniques will be compared based on these characteristics. Also, some simple representative hardening rules will be

introduced. Based on them, a trivial example program will be hardened. That will allow the reader to see how the transformations made by a hardener on a source code can be applied in a concrete way. This example and the corresponding hardening rules will be used through the entire document to illustrate the concepts discussed.

Second, a chapter will be dedicated to the related work. In this part, other hardening projects will be presented along with their testing approach. Also, three popular fault injection tools will be discussed. Their strengths and weaknesses will be detailed.

Fourth, the problem will be fully explained. The context and its constraints will be detailed. Then, the methodology used to make the choice of the solution will be presented. The high-level view of the solution will also be stated.

Fifth, the proposed solution will be concretely explained. This chapter will be cut in section related to the architecture of the developed fault injection testing framework and each part will be presented and illustrated with examples.

Sixth, the validation method will be stated. The first part will contain some benchmarks to evaluate the tool's performance. After, the results of its use on different types of code will be presented. To complete this chapter, some limitations related to the chosen approach will be discussed. Then, conclusions will be drawn based on the objectives, the concrete implementation and the results given by the validation phase.

Finally, some improvements and future work that could be made to the tool or the test approach will be presented.

Note that the code of the tool developed in the context of this thesis and its documentation are available on the following github repository : https://github.com/alvermeylen/fault-injection-master-thesis. Due to confidentiality aspects that will be discussed later, some parts of the source code do not appear on the repository. However, the main features of the tool are functional and can be used.

Chapter 2

Software hardening

2.1 Hardening techniques overview

A program is hardened in order to improve its *dependability*, i.e. its ability to resist to *faults*. When a fault occurs, the program should be able to detect it and act consequently. For example, the application can handle the fault, apply a specific procedure to correct it and continue to run normally or simply stop its execution.

The first hardening techniques were the simplest and were based on specific hardware [3]. The earliest hardened systems were used to ensure the reliability of software executed in space shuttles, because of the greater probability for a cosmic ray to hit the hardware and provoke errors (mainly bit flips). The main drawback of these approaches is the financial cost required to buy more hardware and expensive specialized equipment. Moreover, theses particular pieces are not easily replaced because they are very specialized and expensive, which implies that systems using them become quickly obsolete.

The simplest approach is called *shielding* and consists of protecting electrical systems with some sort of "shields" designed to stop the particles. This technique is clearly the less efficient. First, it adds non negligible weight to the system (which can be an issue in the case of space shuttles). Second, the shields cannot protect from all kinds of particles and some can still go through. Furthermore, the material used to built these shields can itself be a source of radiation and harm health. Finally, some particles might not be completely stopped by the shields but just slowed and still go through [3].

Another hardening technique is to run the program (or at least some of its parts) on several different computers or processors in parallel. Then, the results of these simultaneous executions are compared. Therefore, if the result of a calculation is different from one execution to another, we can conclude that an error happened in at least one of the execution. The main disadvantage of this approach is a greater power consumption.

The EDAC memory is also a hardware hardening technique. It is based on the use of specific hardware pieces that implement error correction codes. It generally allows to correct one-bit errors and detect errors on several bits [3].

To overcome the drawbacks of hardware hardening techniques, it is possible to use software methods to make dependable programs. The main advantages of software hardening are the following : first, the cost is much lower, because no additional or specific hardware is required, which settle the weight problem discussed before. Second, no supplemental electrical power is needed.

An example of software hardening technique is the *N-version programming*, which is based on different equivalent programs, i.e. programs that have the same specifications and output the results in the exact same format. These programs are traditionally implemented by different teams. The system executes each of these programs and compares the results. If the results do not match, the system detected an error. Then, the program can just stop or a voting algorithm can compare the results and guess the correct one [4].

The *recovery blocks* approach is a specific implementation of N-version programming. With this method, specific recovery blocks are created for sensitive computations. Several procedures are implemented to make these computations. A block is like a checkpoint. Before entering it,

the system state is saved. Then the block executes the first procedure and checks the result. If the returned value pass the check, the checkpoint is deactivated and the execution continue. If not, the system returns to its previous state, the recovery block is entered again and the second procedure is executed to make the computation. This continues until the result is correct or all the functions have been tried. [5]

The different software hardening approaches relies on specific rules. These rules define the transformations to apply to a non-robust code in order to harden it. Note that these rules can be used by a *hardener*, which is a specific software used to make programs robust by automatically applying hardening rules to transform the source code.

2.2 Hardening rules

Generally, a hardened program is constructed following some specific *hardening rules*. These rules can be separated in two categories : the ones used to harden the data flow and the ones used for the control flow. They are both complementary.

In the course of this section, a trivial example program will be hardened following the rules described in [2]. These rules and their applications to any source code are simple to understand. They complete the already implemented error detection mechanisms of the system and are applied to high level-code. These rules give good example of software hardening mechanisms and will be used to make validation experiments on the framework developed in the context of this thesis. Note that this approach is mainly aimed at detecting transient faults. However, a big part of permanent faults can also be detected.

The section is organized as follow. First, the trivial example program will be introduced. Second, the concept of data hardening will be explained and the hardening rules from [2] related to the data flow will be introduced. The data flow errors and their classification will also be detailed. Third, the concept, of control flow hardening will be presented along with the specific errors related to control flow. Some popular control flow hardening techniques will be briefly presented and the hardening rules from [2] related to control flow will be explained. Finally, using the introduced rules, our example program will be hardened.

2.2.1 Example program

We will here introduce the trivial example program that will allow us to illustrate the different concepts described in this document. The source code, shown below, represents a simple C procedure. Through this chapter, the code will be transformed to finally become robust against some kinds of faults.

Example procedure

```
1  void procedure(int *res) {
2          int a;
3          int b;
4
5          b = 0;
6          a = b + 1;
7
8          if (a) {
9            b = 1;
10               *res = a;
11         }
12         else {
13               *res = b;
14         }
15 }
```

2.2.2 Data hardening

Data used by programs can be altered. Thus, results of programs' computations using these data will be erroneous. To avoid this kind of incident, techniques of data hardening make use of redundancy, at the level of variables, operations or both. Of course, this redundancy forces to modify the source code. Several possibilities exist : one can duplicate instructions, blocks of code, procedures or even duplicate the entire program. These are the most popular approaches. Another possibility could be the use of special error-correcting bits to detect some types of errors (e.g. parity bit). Obviously, such modifications will affect the execution speed (more instructions need to be executed) and more memory will be needed.

First, the errors affecting the data flow will be presented. Then, a specific duplication technique used to harden the data flow will be detailed. This technique is widely used and will be applied through our example hardening rules.

Errors affecting the data path

Errors that can affect the data path can be found at hardware-level or system-level. These errors can also be classified according to relevant characteristics. Base on these characteristics, we can define some error classes [6].

One can characterize errors by the value of the affected data. Thus, we call **logical errors** inversion of a logical value. For example, a 0 becomes a 1. Contrariwise, **non-logical errors** happen when the affected logical value is outside its definition domain. For example, a 0 becomes something between 0 and 1.

One can also take the *stability* criteria into account. Then one can distinguish between **static errors**, which are stable (e.g. a bit is indefinitely stuck on 1), and **dynamic errors**, which are unstable (e.g. the bit value oscillates between 1 and 0 unexpectedly).

The *duration* of these errors can also be a criteria. Then, errors can be characterized as **hard**, if they are permanent, or as **soft**, if they are temporary (e.g. a value can be modified for one clock cycle only).

Finally, one can differentiate between **single errors**, which affect only one element of the system, and **multiple errors**, which affect several parts of the system.

Based on these classifications, it is now possible to define errors models combining these characteristics.

For hardware-level errors, only two models will be considered here [6] :

- *Single stuck-at* : this is a logical, soft and single error. A typical example would be a logical gate which always give 1 as result.
- *Single bit-flip* : This is a logical, hard and single error. It could be a single bit in memory which suddenly changes its value.

At system-level, errors can affect data used by the system. We will only take these two models into account :

- *Single data error* : this is a logical and single error which corrupt one single data used by the system.
- *Single code error* : This is a logical, and single error which corrupt one single instruction of a program. These errors can be of two types :
  1. The error modifies the operation executed by the instruction but does not affect the program control flow (e.g. an addition becomes a subtraction).
  2. The error affects a jump instruction, which modifies the program's control flow.

High-level instruction duplication

The concept is quite simple. Each variable is duplicated and a consistency check is applied after every read operation to ensure that the value of the read variable is consistent with the value of its replication. Furthermore, the write operations are also duplicated in order to update the value of the duplication and verify that the write operation has not encountered any problems.

One advantage of this method is that it can be entirely automated. Also, because it is applied on high-level instructions, it is not related to hardware. Furthermore, the consistency checks being carried out systematically after every read operation, one can be sure that any error is detected as soon as possible.

Here are the related rules that will be used on our example [2] :

1. Every variable x must be duplicated. x will then become x_1 and x_2.
2. Every write operation performed on x must be transformed on two equivalent write operations on both x_1 and x_2.
3. The two first rules imply that every read operation on x is transformed into two read operations on both x_1 and x_2. After these read operations, a consistency check between x_1 and x_2 must be performed. If the check reveals an error, the error procedure should be triggered.

Then, based on the trivial example found in [2], the simple instruction :

```
1    a = b;
```

is transformed into :

```
1    a0 = b0;
2    a1 = b1;
3    if (b0 != b1) {
4        error_procedure(); //trigger the error procedure
5    }
```

Also note that the *expressions* (e.g. if condition) in which variables are used are considered like read operations on these variables. A consistency check should follow them. Furthermore, the same conclusion can be drawn for the parameters of procedures. Therefore, every parameter must be duplicated, the read operations on these parameters must be followed by a consistency check and the return value must be duplicated too.

2.2.3 Control flow hardening

The control flow of a program represents all the instructions that are executed when a program is run. The order in which these instructions are executed is taken into account. Of course, the order defined by the control flow must be strictly respected to guarantee a correct execution of the program. A fault affecting the control flow is called a *Control Flow Error (CFE)*. Such a fault implies that the processor will execute a different instruction than the one that should be executed. Most of the time CFEs are transient faults that corrupt one or more bit(s) of information. For example, a CFE can be caused by a cosmic ray hitting the processor [3] [6].

Control Flow Graph (CFG)

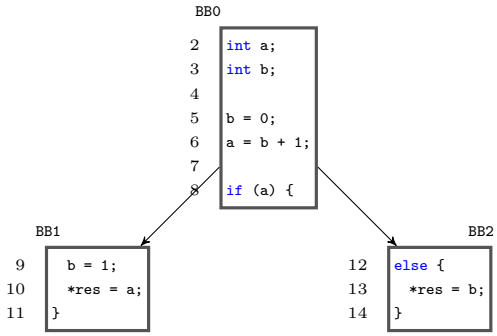
A control flow graph (CFG) is a graph that allows to represent the control flow of a specific program. The first step of the construction of such a graph is to decompose the program into its *Basic Blocks (BB)*. A basic block is a set of consecutive instructions which does not contain any branch. In other words, it is the maximal series of consecutive instructions in which there is no jump instruction (except for the first instruction of the block which can be the result of a

jump or the last one which can provoke the jump). The *basic block body* is defined as the entire basic block without taking the jump implied by the last instruction into account. [6]

Therefore, each node $v_i \in V$ of the CFG represents a basic block and each edge $b_{i,j} \in B$ depicts a branch from v_i to v_j. One can then easily spot the **illegal** branches, i.e. branches that do not appear in the CFG. However, The program can also take a **wrong** branch, i.e. the program execution take a branch $b_{i,k} \in B$ instead of $b_{i,j} \in B$. These two special kinds of branches depict CFEs. [6]

As an example, we will use the trivial code of our example procedure introduce above and give its corresponding Control Flow Graph :

```
1   void procedure(int *res) {
2     int a;
3     int b;
4
5     b = 0;
6     a = b + 1;
7
8     if (a) {
9       b = 1;
10      *res = a;
11    }
12    else {
13      *res = b;
14    }
15  }
```



Control Flow Errors

The control flow errors can be divided into two main categories : the intra-blocks errors, i.e. the errors where the branches does not respect the blocks' boundaries, and the inter-blocks errors which are the ones in which source and/or destination of branches is (are) not correct. These faults can be divided in several types [6] :

- Type 1 : inter-block errors caused by an illegal branch.
- Type 2 : inter-block errors caused by a wrong branch.
- Type 3 : intra-block errors where the final jump of a BB lead to the body of another BB.
- Type 4 : intra-block errors where a jump happens from the body of a BB leading to a random instruction from the body of another BB.
- Type 5 : intra-block errors where a jump happens from a random instruction of a BB body to another random instruction of the body of the same BB.

Control Flow Check (CFC)

There exists several techniques to check control flow and so detect CFEs when they happens. In this document, we will only speak about signature checking approaches. The principle of such methods is simple : a signature is associated to the structure of the program (e.g. a unique signature for each basic block) at compile-time (or, in any case, before execution). During execution this signature is dynamically computed. If the value of the signature computed during execution is different than the value assigned during compilation, one can deduce that a CFE happened. [6]

Several concrete control flow checking methods are based on this principle. We will briefly present on table 2.1 some characteristics of some of the most popular CFC approaches [6].

As a concrete example, here are the hardening rules from [2] that will be applied on our example procedure :

4. An ID k_i must be assigned to each basic block of the program.

5. a global execution check flag variable (`ecf`) must be defined. Then, the first instruction of each basic block b_i must be an update operation $ecf = b_i$. The last instruction of every basic block b_i must be a test $ecf == b_i$. Note that this approach has two main limitations : first, a jump between two instructions of a same basic block cannot be detected ; second, a jump to the first line of any basic block cannot be detected.

6. For every conditional statements (e.g. if statements, loops, etc.) the test is performed a second time at the beginning of the basic blocks corresponding to the "true" and "false" branch. An error is detected if the two tests return different values.

7. An ID k_j is assigned to every procedure.

8. Just before returning a value or after the last instruction if no return are performed, a statement from the function or procedure updates `ecf` to k_j. A consistency check on `ecf` is performed after the function/procedure call.

2.2.4 Hardened example

Now, using the rules defined before, we will harden our example code. Also, two other files, `hardening.c` and `hardening.h`, will be created to support the hardening features added in the code. These features are : a `check` function in order to perform the consistency checks required by some hardening rules, the `ecf` variable and an `error_flag`. This last variable will be assigned to 0 at the beginning of the program execution and will be incremented by one by the `check` function each time the consistency check fails. We will then be able to see if any error happened during the execution of the hardened procedure by verifying the value of `error_flag`.

Here are the `hardening.h` and `hardening.c` files :

hardening.h

```
1  #ifndef HARDENING_H
2  #define HARDENING_H
3
4  int ecf;
5  int error_flag;
6
7  void check(int assertion);
8
9  #endif /* HARDENING_H */
```

hardening.c

```
1  #include "hardening.h"
2
3  void check(int assertion) {
4          if (! assertion) {
5                  error_flag++;
6          }
7  }
```

Below are the hardened version of the procedure presented before :

Hardened example procedure

```
1  #include "hardening.h"
2
3  void procedure(int *res0, int *res1) {
4          //RULE 4 & 5
5          ecf = 0;
6
7          //RULE 1
8          int a0, a1;
9          int b0, b1;
10
11         //RULE 2
12         b0 = 0;
13         b1 = 0;
14         a0 = b0 + 1;
15         a1 = b1 + 1;
16
17         //RULE 3
18         check(b0 == b1);
```

```
19
20          //RULE 5
21          check(efc == 0)
22
23          if (a0) {
24                  //RULE 4 & 5
25                  ecf = 1;
26
27                  //RULE 3 - expression
28                  check(a0 == a1);
29
30                  //RULE 6
31                  check(a0);
32
33                  //RULE 2
34                  b0 = 1
35                  b1 = 1
36
37                  //RULE 2
38                  *res0 = a0;
39                  *res1 = a1;
40
41                  //RULE 3
42                  check(a0 == a1);
43
44                  //RULE 5
45                  check(ecf == 1);
46          }
47          else {
48                  //RULE 4 & 5
49                  ecf = 2;
50
51                  //RULE 3 - expression
52                  check(a0 == a1);
53
54                  //RULE 6
55                  check(!a0);
56
57                  //RULE 2
58                  *res0 = b0;
59                  *res1 = b1;
60
61                  //RULE 3
62                  check(b0 == b1);
63
64                  //RULE 5
65                  check(ecf == 2);
66          }
67  }
```

As you can see, each modification of the `procedure` function has been justified with the related hardening rule.

2.3 Tests

Hardened programs need to be tested as any program. However, some specific tests should be performed to verify the correctness of the errors detection mechanisms implemented through the hardening rules. To achieve so, two kinds of tests should be used on hardened programs.

First, if the program has been hardened based on non-hardened counterparts, functional tests should be used to verify that the original program and its hardened version are functionally equivalent.

Second, the program will be tested trough simple unit tests. These tests have two roles. First, they complete the functional tests by checking the correctness of the value returned by the tested function. Second, they will allow to challenge the hardening rules implemented by the robust program by checking the value of the error flag. Note that we suppose that an error flag value equal to zero means that no error were detected. Four different results can be observed and are reported on table 2.2.

Only the first case report a good execution of the program. Indeed, the returned value is correct and the error flag has not been modified.

In the second and third cases, the error flag has been modified. As it is a simple test situation, no error should have happened. However, the error detection mechanisms have been triggered. That reflect an error in the hardening rules.

The last case is also a failed test. The error flag was not modified, however the returned value is not the one expected. That reflect a functional error as the tested procedure does not work as intended.

Here is an example of unit test that could be applied on our example procedure :

Unit test

```
1  #include "hardening.h"
2
3  extern void procedure(int *res0, int *res1);
4
5  int main(char **args) {
6         error_flag = 0;
7         int res0, res1;
8         procedure(&res0, &res1);
9
10   return res0 != 1 || error_flag;
11 }
```

As you can see, the returned value of this unit test is quite unconventional. If the test passed, the program return zero (false) ; else, it return a value different from zero. This convention is commonly used in hardening tests using error flags because the default value of this flag is usually zero. This convention will be used in the rest of this document.

CFC technique	**Advantages**	**drawbacks**
BEEC and ECI	• Manage CFEs of type 1 to 4	• Don't manage type 5 CFEs
ARC and VASC	• Good performances • Manage CFEs of type 1, 2 and 4 • The majority of type 3 CFEs are detected	• Don't manage all type 3 CFEs • Don't manage type 5 CFEs • Latency in error detection
ECCA-HL and ECCA-IL	• Manage CFEs of type 1, 3 and 4 • Good performances for ECCA-IL	• Don't manage CFEs of type 2 and 5 • Bad performances for ECCA-HL
Plain inter-bloc errors detection	• Manage CFEs of type 2, 3 and 4 • Simple approach	• Don't manage CFEs of type 1 and 5
CFCSS	• The majority of CFEs of type 1, 3 and 4 are detected • Good performances	• Don't manage CFEs of type 2 and 5 • Not all type 1, 3 and 4 CFEs are detected
ACFC	• Good performances • Partially manage CFEs of type 1, 3 and 4	• Don't manage CFEs of type 2 and 5 • Partially manage CFEs of type 1, 3 and 4
YACCA	• Manage CFEs of type 1, 2, 3 and 4 • Good performances	• Don't manage type 5 CFEs • Big latency in error detection
SIED	• Manage all kinds of CFEs • CFEs that happens in special instructions for error detection mechanisms are detected	• Bad performances (lots of added instructions)

Table 2.1: some popular control flow hardening techniques

Returned value	**Error flag**	**Test result**	**Observation**
TRUE	0	Passed	No error detection mechanism have been triggered and the result value is good.
FALSE	!= 0	Failed	The result value is not the one expected and the error detection mechanisms have been triggered
TRUE	!= 0	Failed	The result value is the one expected but the error detection mechanisms have been triggered.
FALSE	0	Failed	The result is not the one expected and no error detection mechanisms has been triggered.

Table 2.2: interpretation of a unit test on a hardened program

Chapter 3

Fault injection

To test a hardened program, it is necessary to verify against which types of error it is robust. It will allow to check its reliability. To do so, an approach is to inject faults into the system to see how the program under test will react. These faults can be of several types. It can be, for example, the modification of the value of a variable through bits flips or a sudden jump from one instruction to another. Thanks to a test set based on these kinds of faults, the errors that are not managed by a robust program can be determined.

As detailed below, there are several techniques to perform fault injection. Each of them has its particularities [7] [8].

3.1 At hardware level

The simplest way to inject faults is to directly affect the hardware pieces of the system on which the tested program is executed. This is the most basic technique but also the most realistic one. However, it has two major disadvantages. The first is the high cost to acquire specific hardware allowing to affect the system (e.g. through radiations or electrical field). The second is the risk to damage the system's hardware on which the injections are performed. Its main advantage is the execution speed. As it does not touch the software, the execution speed of the tested program will remain the same when a software fault injection approach could make the execution slower.

There are two different types of hardware fault injection methods. They are described below.

3.1.1 Injection with contact

The injection techniques that require contact rely on specific material that needs to be physically attached to the hardware pieces to affect.

The first solution is to use "active probes". These probes will be placed on the circuit and will alter the electrical signals via electromagnetic interference. The used of such probe is represented on figure 3.1 from [7].

The second way is to plug a special socket between the motherboard and another hardware piece (like memory, processor, I/O device, etc.) . The current traveling through this socket will then be altered before going to its destination. This can be visualized on figure 3.2 from [7].

3.1.2 Injection without contact

These types of approaches require specific material in order to produce heavy-ion radiations or magnetic fields near the hardware on which the faults need to be injected. The electrical current going through the system will then be altered. This is the most realistic kind of fault injection approach because it models the real conditions to which a system could be confronted in the real world. However, the generated faults are totally random. Thereby, the tester cannot reproduce exactly experiences or results.

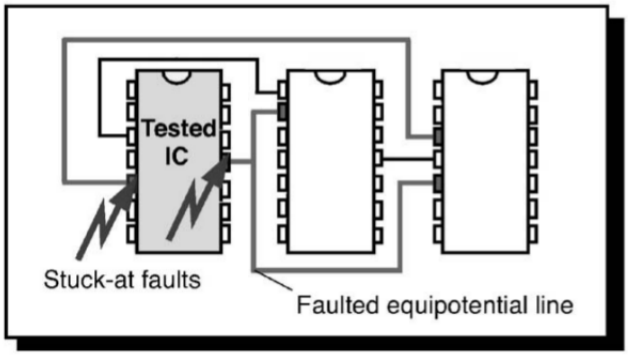


Figure 3.1: Fault injection with probe [7]

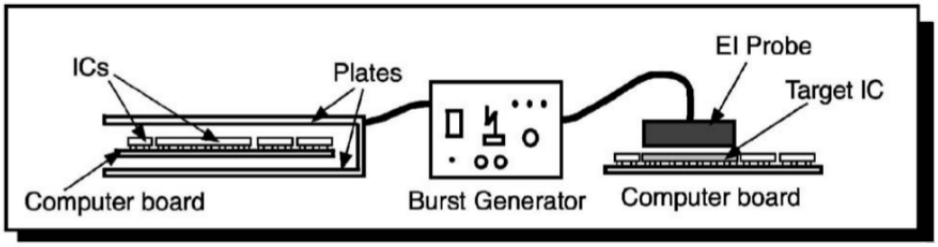


Figure 3.2: Fault injection with socket [7]

3.2 At software level

The software-level fault injections can be performed directly in the application to test or between the application and the operating system. The main advantage of this type of approach is that the costs are minimal because no specific hardware is required. However, it slows the execution speed. Also, some techniques modify the software source code. The tested program is then not exactly the original application. Furthermore, these techniques cannot exactly reproduce some physical effects that could provoke a fault in a real situation. For example, an electrical signal cannot be altered in a software way.

There are two main approaches. The first one is to design a specific fault injector that will affect the program to test during its execution. The second one is to make the program run on a simulated environment that can be corrupted during the test run.

3.2.1 Software fault injection

As for the hardware-level methods, two types of software fault injection techniques can be distinguished.

Injection at compile-time

With this approach, the program to test is modified in order to simulate the fault. This modification can take place at the source-code or compiled-code level, as represented on figure 3.3. This approach is less flexible because the faults are "hard-coded". Indeed, the same fault is injected at every execution and the simulation of another fault require a new modification of the original software.

The *mutation testing* method can be used to make fault injection at compile time. It will be described in the next chapter.

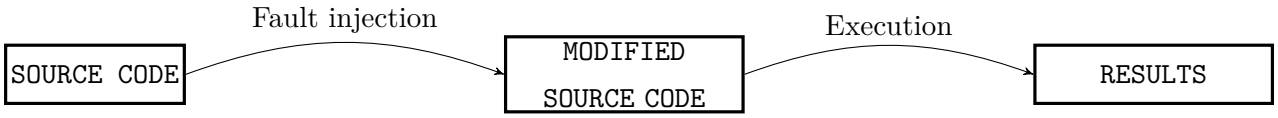


Figure 3.3: Compile-time fault injection process

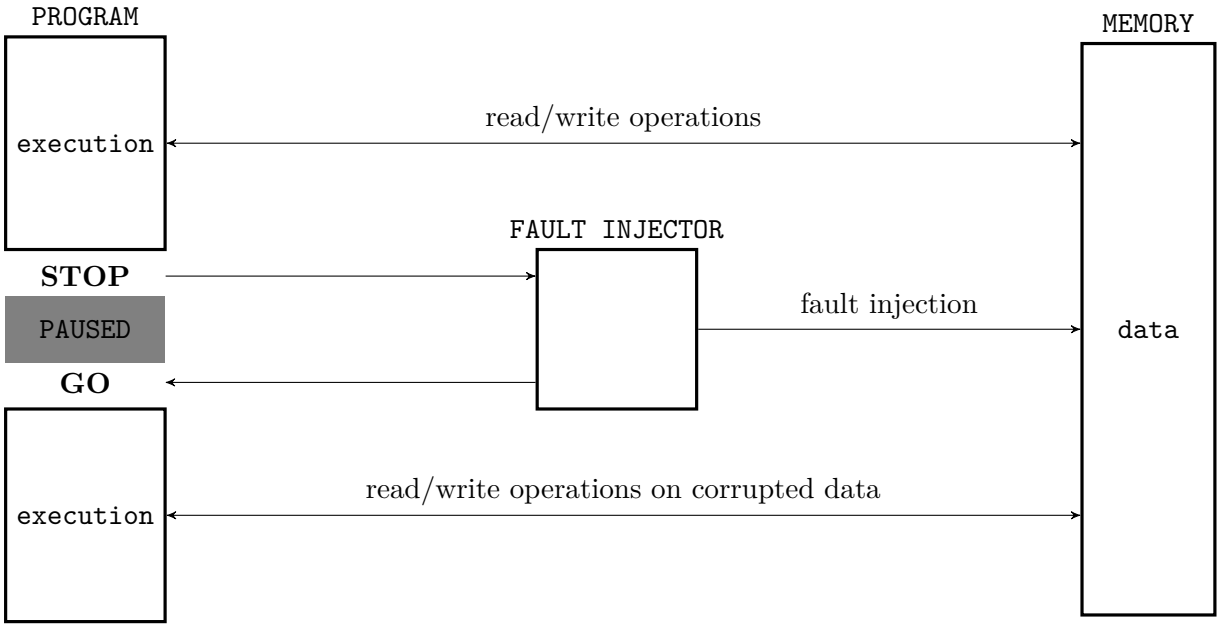


Figure 3.4: Run-time fault injection process

Injection at run-time

There also are some techniques to inject faults during run-time :

- *Time-out* : It is the simplest approach. A fault is injected after a defined time-out has expired. This time-out can be defined in a software or hardware way.
- *Exceptions* : In this case, a fault is injected after a specific exception has been triggered. Note that an exception is an hardware mechanism. It can, for example, be the access to a certain piece of hardware (hard disk, specific memory zone, etc.).
- *Traps* : With this technique, the injection is performed when a trap has been triggered. A trap is a specific instruction that causes the fault injector to be called. The principle is the same as for the exceptions, except that a trap is a software mechanism.
- *Code insertion* : The principle is the same as for the compile-time injection techniques, except that the instructions are added during execution. The difference with the traps approach is that when a trap is triggered, a, external injector is called, whereas here, the program makes the injection itself.

The general concept is represented on figure 3.4.

3.2.2 Simulation

This approach is based on a simulator that will emulate the hardware on which the program will be executed. The faults can then be injected at the emulated hardware level. The main advantage is that the faults can be injected at all levels, i.e. every emulated location can be

altered. The other software injection techniques do not allow that. When a simulation method is used, the tester needs to be sure that the emulated model represents a real system in a relevant way. Once the first model has been created, the simulator can be altered to model errors. Also, modules can be used to dynamically corrupt some part of the simulated system. Another possibility is to use the simulator's engine language to force faults. For this purpose, the VHDL language is a popular tool. It is a description language that allows to model electronic systems. Example of such a fault injector can be found in [9]. Also, the general concept is represented in figure 3.5 from [6].

This approach is the most realistic of the software fault injection techniques. However, the development of such a simulator is a really long and complex work. Moreover, these kinds of injector often rely on randomness. Indeed, the simulated hardware is altered in the same way the hardware techniques inject faults. The simulated system is corrupted but not the program directly.

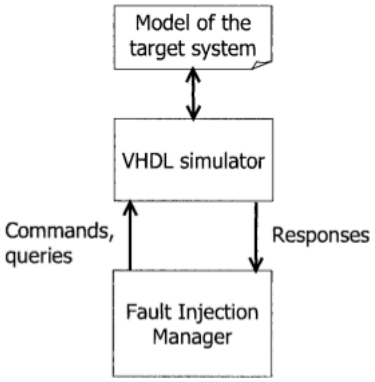


Figure 3.5: Simulated fault injection solution [6]

3.3 Comparison criteria

We will explain here the different criteria that can be used to evaluate and compare fault injection techniques. After describing these criteria, we will apply them to the main approaches in order to be able to evaluate them.

3.3.1 Fault representativeness

When it comes to evaluate a fault injection technique, several criteria have to be taken into account. These criteria will be detailed in the next section. First of all, we have to define the most important property of fault injection : the fault representativeness. This characteristic represents the "realism" of the model used by the injector to inject faults. Naturally, the strategy adopted by the injector must correspond to what can happen in reality as much as possible [7].

To have a high level of representativeness, two approaches can be considered. First, one can based his strategy on a set of faults inspired from real-world situations. The injector will then be implemented in a way such as these real faults are modeled as close to the reality as possible when it comes to choose when and where to inject. Although it is theoretically the best technique, it is not always possible to collect sufficient pertinent information about real faults. Thenceforth, an alternative is merely to use several separate techniques in parallel. Two conclusions are then possible. The first case is when several injection methods are *equivalent*,

that is to say they lead to the same results and allow to identify the same limitations on the tested program. In this case, one must choose between the considered approaches based on the criteria defined in the next section. The other case is when the fault injection techniques are said to be *complementary*. In this case, it is favorable to use the distinct techniques in conjunction [7].

3.3.2 Comparison criteria

We will define here the different criteria traditionally used to compare or evaluate a fault injection approach or tool [7] [8].

Reachability

This characteristic represents the ability of the fault injection technique to reach physical area where a fault can occur and effectively inject one. A fault can take place in a lot of different places like memory, in some processor area during a computation, at the register level, at the electrical signal level, in a logical gate, etc. Of course, if a fault injection approach can access more physical zones, it can inject at more levels which will allow it to model a larger set of real faults.

Controllability

When speaking about controllability, two things need to be taken into account. First, the physical location where the fault will be injected (the variable that will be modified, the statement that will be affected, etc.). Second, the time when the injection will be performed. More controllability implies that the system can be tested in a more precise way which will allow to simulate more specific errors. That can enable to monitor how the system will react to a particular error or to model faults inspired by real-world errors.

Repeatability

Repeatability is directly linked to the previously cited characteristic. This criteria is very important in this context because it represents the capability to reproduce the exact same experiment several times. Indeed, if one wants to challenge or improve the dependability of a system in order to resist to a specific kind of fault, it is primordial to be able to repeat an experiment after the program has been modified or improved. It is also significant to allow one to inject a particular kind of fault.

Reproducibility

Unlike the repeatability, the reproducibility define the capability to reproduce the same results (and not the same experiment). Of course, this characteristic is strongly related to the previous one. However, a same experiment can give different results and same results can be obtained through different experiments.

Nonintrusiveness

We characterize as intrusive a fault injector that modifies the behavior of the system on which the tested program is executed. When faults are injected to test a specific program, we want these faults to affect only this program and not the system on which it is executed. The nonintrusiveness represents the capability to corrupt only the program under test and modify its behavior without having any effect on another part of the system.

Time measurement

During a fault injection test, it could be interesting to be able to measure the time between the actual fault injection and the fault detection by the program under test. Also, a measurement of the time lapse between the injection and the time on which the fault has an actual effect on the program behavior could be relevant. A more precise measurement of these time intervals allows the tester to draw better conclusions.

Efficacy

In this context, a fault injection method is considered efficient if it leads to the least possible number of non significant experiments.

Costs

The costs related to the establishment of such tests have also to be taken into account. The goal is to find the good balance between the costs and the quality of the approach(es) considered, which relies on several factors. If the software to test will have a sensitive influence (put lives in danger or risk a great financial lost for example) the cost will have less significant importance.

Note that the cost can be measured in different ways, depending on context. It can be calculated in money, development time or other.

in addition to the costs of development and set up, risks to damage the hardware used to perform the tests can be taken into account too. While there is no such risks with software injection methods, physical injection approaches can sometimes harm the material on which it is used. Note that the risk is lower with physical injections methods without contact.

3.3.3 Comparison

Based on information given in [7] and [8], the table 3.1 gives information about the criteria detailed previously for different fault injection methods.

	Hardware with contact	Hardware without contact	Software by simulation	Injection at compile-time	Injection at run-time
Reachability	High	Medium	Medium	Low	Low
Controllability (time)	Low/medium	None	Medium	High	High
Controllability (location)	High	Low	High	High	High
Repeatability	High	Very low	High	High	High
Reproducibility	High	Variable	High	High	High
Nonintrusiveness	Medium	Low	Medium/high	High	High
Time measurement	Medium	Low	Medium/high	High	High
Efficacy	High	High	High	Low/medium	Medium/high
Cost	High	High	Medium	Low	Low
representativeness	High	High	Medium	Low	Low

Table 3.1: fault injection criteria for each kind of approach

As we can see, Five main differences between hardware and software approaches can be identified. First, the cost is really high for hardware techniques compared to software ones. Second, the controllability with relation to time is clearly better for software methods. Third, the reachability is higher for hardware techniques. Fourth, the hardware approaches are more

efficient than the software ones. Finally, The representativeness of the injected faults is higher for the hardware techniques.

Also, comparing the software techniques, the simulation approach seems to be less attractive than the two others, which seems pretty equivalent.

Chapter 4

Mutation testing

Mutation testing is a technique that can be used at compile-time to inject faults. We will first explain the concept of mutation testing and then briefly discuss its weaknesses [10]. Finally, we will discuss its use as a software implemented fault injection technique.

4.1 The concept

This testing technique is not intended to prove the correctness of a program but to find the faults that the tested program cannot manage (yet). Another use of this technique is to evaluate and improve the quality of a given test set on a program.

The principle is to modify the source code of a program in order to make it intentionally faulty. A modified version of the original program is called a *mutant*. Once all the mutants have been generated, one can run them in order to see and analyze the effects of the simulated faults on the original program. Note that this approach require a predetermined set of tests.

Mutation testing can be used for unit tests, integration tests and even tests aimed at the architecture or specifications of a program. In this last case, mutations need to be applied not on the program itself but on a representation of it, like a state machine or a Petri net. Note that all the errors that could possibly affect a program cannot be modeled by a mutant.

The mutation testing approach rely on two hypothesis :

- Competent Programmer Hypothesis (CPH) : this assumes that the programmer is competent, which means that the faults he made are simple. Therefore, the mutants must only contain simple faults.
- Coupling Effect Hypothesis (CEH) : this assumes that most of the complex faults are simply the result of a combination of several simple faults. Thenceforward, only simple faults are to be considered in mutants because, once detected, they will allow to detect the complex faults of which they are the source.

4.1.1 Mutation operators

Mutation operators are used to create mutants from the original source code. They are simply rules that allows to apply the desired mutation. An example of such an operator could be to transform all '+' symbols in '-'.

Note that a mutation operator affects only one syntactic entity of a program. Mutations operators should have the following characteristics [11] :

- They introduce only simple faults.
- The produced mutant is syntactically correct, i.e. the mutant can be compiled and executed.
- Every operator is correctly defined and classified. This implies that each operator belongs to a specific class which is defined according to the effect of its corresponding mutation.

This can be, for example, syntax-directed. The naming convention of the mutation operator is based on the chosen classification. This makes the task easier for a tester who wants to use these operators.

As an example, we will apply a mutation operator which transform '+' symbols in '-' on the trivial procedure introduce in the previous chapter :

Original example procedure

```
1  void procedure(int *res) {
2          int a;
3          int b;
4
5          b = 0;
6          a = b + 1;
7
8          if (a) {
9            b = 1;
10                *res = a;
11         }
12         else {
13                *res = b;
14         }
15 }
```

Mutant version of the example procedure

```
1  void procedure(int *res) {
2          int a;
3          int b;
4
5          b = 0;
6          a = b - 1; //Mutation
7
8          if (a) {
9            b = 1;
10                *res = a;
11         }
12         else {
13                *res = b;
14         }
15 }
```

As you can see, the symbol used at the sixth line have changed.

When a test is performed on a mutant, we need to check that the result is different from the one the test would have returned if executed on the original program. If it is the case, the mutant is said to have been "killed" ; if not, the mutant is "alive". A typical test scenario is describe hereafter and graphically represented on figure 4.1 from [10].

1. A test set T is defined.
2. T is performed on the original program p. All the tests must pass ; if not, p must be corrected.
3. T is performed on every mutant p'.
4. If after all the tests have been performed, some mutants are still alive, other tests are added to T in order to kill them.
5. The newly added tests are performed until every mutant has been killed.

Therefore, one could think that the result of a test performed on a specific mutant can be concluded in two ways : either the mutant is faulty (implying that the original program is correct), or the original program is faulty. However, some mutants can never be killed because, even if they are syntactically different of the original program, they are functionally equivalent. Based on that, we can define the *mutation score* (MS) which is the ratio between the number of killed mutants on the number of mutants which are not equivalent to the original program.

$$MS = \frac{killed\ mutants}{mutants\ non-equivalents\ to\ p}$$

A mutation score close to one means a good test set.

4.2 Weaknesses

The mutation testing method has two main drawbacks : the efficiency and the computation of the mutation score.

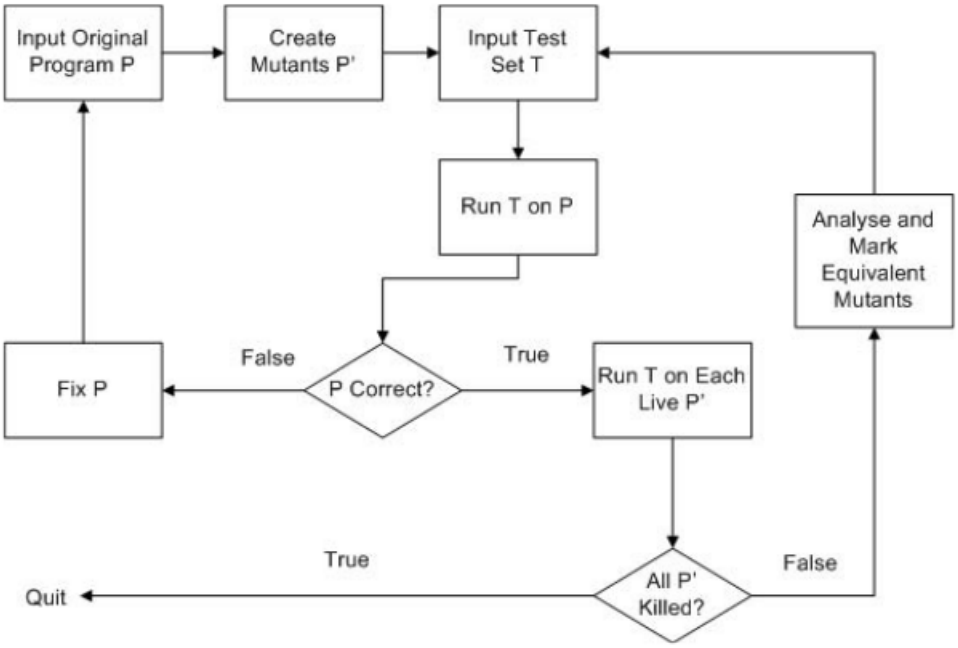


Figure 4.1: Mutation testing process [10]

4.2.1 Efficiency

Executing the entire test set (for each mutant) is often costly in terms of computation time. Two approaches can be considered to reduce it : reduce the number of mutants and reduce the execution cost.

Reduce the number of mutants

A high number of mutants logically implies a long time to execute the test set. There are several techniques to reduce the number of mutants without affecting too much the quality of the test set. Some of these methods are explained below.

First, the *mutation sampling* method allows to reduce the number of mutants by selecting only a sample from all the generated mutants. Experiments have shown that taking a sample of only 10% of all the generated mutants only reduce the test set efficiency by 16% [12] [13]. Generally, the sample size is based on a specific probability computation.

Second, the *mutant clustering* approach can be used. The first step is to generate all the mutants. Then we consider that, if two mutants can be killed by a same test form the test set, they belong to the same cluster. Based on the hypothesis that all mutants from a same cluster are equivalent, we only need to use one mutant from each cluster.

Third, the *selective mutation* approach allows to reduce the mutation operators used to produce mutants. We can, for example, omit the operators that generate the larger number of mutants or intelligently select the used operators. In this last case, we speak about "constraint mutation".

Finally, *higher order mutations* can be used. In general, only one mutation operator is used to generate one mutant. With this approach, we use several operators by mutant.

Reduce the execution cost

We will briefly define here two methods that can be used to reduce the execution time of the entire test set on all mutants.

The first distinguishes between *strong*, *weak* and *firm* mutations. In the case of strong mutations (which is mainly used), we consider that a mutant is killed if its final output is different from the final output of the original program. This implies that the entire mutant has to be executed. Another approach can be used in which the program is cut into blocks. As a mutation only affects one block, we only need to execute that block and not all the program. The mutant is killed if the execution of the mutated block is different from the original one. Firm mutation lies in between. A location is chosen between the affected block and the end of the program when we can safely conclude if a mutation had an impact on the program.

The second one consists of run time optimization techniques. First, in the case of interpreted languages, the source code can be transformed into an intermediate representation which is more flexible and allows to make mutation easier. Then, for compiled languages, instead of compiling the original program plus every mutant, we can choose to only compile the original source code. Then, a patch is generated for each mutant. The mutation is then simply to apply the patch on the compiled program. Another method is to only compile the original program and a "meta-mutant" which has the capacity to represent every mutant.

4.2.2 Computation of the mutation score

To evaluate the tests set and so, indirectly, the original program, we use the mutation score as indicator. Although, to compute the mutation score, we need to be able to compare the mutants to the original program in order to see if they are functionally equivalent. However, this problem is undecidable. It is then not possible to automatically compute the exact mutation score.

However, some approaches are used to detect equivalent mutants. One is to use the optimization techniques provided by the compiler. Indeed, when a source code is compiled, the compiler use different types of optimization methods to make the program the most efficient possible. We can say that, if two programs are exactly the same once they have been optimized by the compiler, we can safely conclude that they are functionally equivalent. This method allows to detect some equivalent mutants.

Other approaches are used to make the task easier for humans. Indeed, if one wants to detect equivalent mutants, he can do it manually. The program slicing technique, for example, is used to decompose the source code into slices, based on the outputs provided by the mutants. This filtering can help a human recognize equivalent mutants.

4.3 Mutation technique and fault injection

The mutation testing technique can be considered in order to test the robust capabilities of hardened programs. Indeed, the mutation operators applied could have been chosen to represents the different kinds of fault that need to be simulated. Such a technique is a special kind of injection at compile time.

4.3.1 Evaluation

As the mutation testing technique used for fault-injection could be classified as a injection at compile time approach, it have the same global advantages and drawbacks. However, even if this technique could be used in the context of this thesis to simulate faults into a hardened program, it suffers from a few weaknesses.

First, as we have seen, the number of generated mutants can rapidly explode. The tests will then take a lot of time to execute. Moreover, each of these tests will be a complete program that will need to be executed and stored.

Second, the (insolvable) problem of equivalent mutants risks to lead to a lot of not relevant test cases. If a mutant is equivalent to the original program, it will produce a *false positive*, i.e. the faults will not be detected because the mutation does not simulate an error. The efficacy is then quite low.

Finally, a major drawback is the fact that the mutants are tested, not the original program. As the source code has been modified, it is actually a different program that is tested.

On the other hand, as the mutation operators could be precisely defined, such an approach would have a high controllability, repeatability and reproducibility.

Chapter 5

Related work

We will describe here some existing solutions and tools to inject faults into systems in order to test hardened software. First, we will present some software hardening projects and detail the test approach used. After that, we will list some of the most popular fault injection tools and briefly explain their characteristics.

5.1 The TIMA Lab. hardener prototype

5.1.1 Context

In the research report published by the TIMA Laboratory [14], an approach to modify software in such a way that they became tolerant to transient faults is detailed. The proposed technique is based on a set of rules whose goal is to transform high-level code by adding data and code redundancy in order to produce an equivalent software that can handle some sort of faults. These hardening rules can be divided in two categories : the rules that target faults affecting data and the ones targeting faults affecting code. The advantage of this method is that it can be fully automated. As we will see, these rules are very similar to the ones used in our example procedure.

The first kind of rules simply duplicates every variable. Each time a write operation is applied to a variable, its duplicated copy go through the same operation. Also, whenever a read operation is performed on a variable, the consistency between this variable and its duplicate is verified.

The second type of rules is based on two ideas. First, some rules are designed to duplicate every operation in order to assure the respect of the normal execution path. Since all instructions involving operations on variables are duplicated due to rules protecting the data, these rules are intended to duplicate the jump instructions (e.g. the condition of an if instruction is repeated twice). Secondly, some rules are meant to protect the program from control flow error. To achieve that, the program is divided into its basic blocks and a flag variable is introduced. Each time the execution reach the beginning of a basic bloc, the flag is set to the value of the id of the basic bloc. Then, the last instruction of a block test if the value of the flag is coherent with the id of the basic block.

5.1.2 The test approach

In order to test and validate this technique, the TIMA Laboratory developed a prototype of a hardener using the rules presented above. This hardener receives a correct C source code as input and output a equivalent program tolerant to transient faults.

This prototype was then used to demonstrate the correctness of the hardening approach. To do so, two techniques of fault injection were used on it, a software one and a physical one, each of them involving only bit flips.

Software fault injection

Most of the experiments were performed via this approach. To do so, a prototypical board developed by TIMA Laboratory was designed to make fault injections easier. Based on this board, a specific fault injector was implemented. Then, both the program under test and the injector were loaded on the board. While the tested application was running, the injector had to wait a random amount of time, then perform a single bit flip on a random bit from a random chosen address contained in the memory area used by the tested program. After each fault injection, the program behavior was recorded to be analyzed by the testers.

Four types of faults were identified : the *effect-less* ones, that doesn't alter the program execution ; the *software* and *hardware detected* ones ; the *no answer* faults, that cause the program to never give an output (e.g. because of an infinite loop) ; the *wrong answer* faults, that cause the program to deliver an incorrect output, which are the faults against whose the program must be hardened.

Three different programs were used to perform the tests : a multiplication of two 10 x 10 *matrix* of integers ; a *bubbleSort* and a *quickSort*, both on an array of ten integers. The same methodology was applied for the three applications. First, the programs were given to the hardener in order to get the hardened versions. Secondly, the overhead in terms of code size and execution time were computed. Finally, a fault injection experiment were performed on the two versions of each application.

After the tests executions, the number of faults of each type were compared for the two runs. The conclusions are that for the faults affecting the code, 45.5% of the faults injected leaded to wrong answers for the original versions while this situation was avoided for more than 90% of the injections performed on the hardened programs. Regarding the faults affecting the data, none of the modified versions gave wrong answer while the original programs' output was incorrect in 80% of cases.

Physical fault injection

Another test approach was also used by the TIMA Laboratory, it is the heavy-ion radiation. To perform the experiment, a part of the memory of the prototypical board used for the software fault injections was exposed to heavy-ion radiations. Then, particles hit the memory and induced bit flips in it at random time and random location.

Due to the complexity of performing such an experiment, only the *matrix* program was tested. The results obtained from the two injection techniques were compared. They are reported in the table in figure 5.1. However, keep in mind that, the heavy-ion radiation experiment was less complete and then not entirely comparable to the software injection one.

FAULT INJECTION VS. RADIATION TEST ING

	Radiation test	Fault injection
# upsets	4,377	4,488
No effect	2,136	1,856
SW detection	1,920	2,312
HW detection	204	256
Wrong answer	93	12
No answer	24	52

Figure 5.1: Comparison of the two fault injection experiments on the TIMA prototype

5.2 The MAintainable Real-Time System (MARS)

5.2.1 The project

As explained in [15], the MARS system is intended to deal with critical distributed applications. This real-time system is composed of autonomous nodes that communicate thanks to exchanges of messages through a real-time network. Each node must have a fail-silent behavior, i.e. the system will never deliver an incorrect output. To do so, when a node experiences a failure, it detects it and simply shuts down in order to confine the failure and preclude its propagation.

To achieve fault tolerance, the MARS system is composed of specific hardware and an operating system designed to handle errors. The hardware errors detection mechanisms are mainly intended to detect permanent errors, the software errors detection mechanisms are primarily targeted for the detection of transient faults that can cause data corruptions or control flow errors. These mechanisms can be decomposed in three levels [16] :

- Hardware level : handles some data errors and errors caused by the environment.
- System level : handles data and control flow errors and behavioral errors.
- Application level : handles errors in the transfer or processing of the data (e.g. CRC code are used to check the integrity of messages after they have been received by a node).

5.2.2 Evaluation of error detection mechanisms

A SoftWare Implemented Fault Injection (SWIFI) approach has been developed to test the MARS system. To be more explicit, two different SWIFI techniques were used : one based on the bit-flip fault model and another to specifically test the application level fault detection mechanisms [16].

The first approach is the one that could be the most adapted to our problem. The MARS testers have chosen a pre-runtime injection approach. This choice is relevant in this case because it is the system that need to be tested. So, the bit-flip faults are injected by modifying the code of applications that will be run in the MARS system. The advantage is that the system is not modified by the fault injections. To perform a fault injection experiment on the MARS system, the tester have to write a high-level description of the experiment that will be run (as some kind of script), containing, among other, the location where the faults will be injected, the number of faults, their type, their distribution and the number of runs needed for the experiment. Then, this "high-level script" is given to a conversion tool which will generate "low-level scripts" for each experimental run. Afterwards, these scripts will be loaded into the MARS system and the fault injection experiment will begin. During the tests runs, a supervision program will monitor the system behavior in order to collect relevant data that will be recorded to be analyzed by the tester. The main characteristics of the MARS experiments were the following :

- Three different configurations of the system were tested : one where all the application level error detection mechanisms were disabled and two others where only a part of these mechanisms were enabled.
- Faults were injected into the code or the data segment of the corrupted applications.
- Only single bit-flips were used.
- The locations of the injected faults were random.
- From one to ten faults were simultaneously injected during a test run.
- Not all injected faults was effective. Some of them could not be reached due to the application execution path.

The main disadvantage of such a solution is that it is not automated and require a tester to write a description of the fault injections he wants to perform. Also, the fact that the faults locations are chosen randomly does not allow the experiments to be fully controllable [7].

The second approach is very specific to the MARS system and not interesting to be fully detailed here. To summarize, these experiments were based on a corruption of the messages exchanged between the nodes in order to alter either their content or the error detection mechanisms, in such a way to model worst case scenarios.

Note also that the MARS system were used to test and compare three physical fault injection techniques (pin-level, heavy-ion radiation and electromagnetic interference) [17].

5.3 Gpu-qin

Gpu-qin is a tool implementing a particular fault injection methodology to test the resilience of GPUs to errors [18]. Nowadays, GPUs are used for general-purpose applications (GPGPU). If the GPU is used for graphical purpose only, a fault can seldom be really problematic. It will result in a few wrong pixels. However, technologies like CUDA allow to make use of the GPU for *general-purpose processing*. In this case, a fault can have serious implications. In this context, Gpu-qin was developed in order to test the robustness of GPGPU applications. The project focus on GPGPU applications specifically implemented on NVIDIA architectures using the CUDA technology, which provides quite popular programming model and toolset.

One specific problem related to GPGPU applications is that these applications make a huge use of parallelism. Thereby, a lot of threads are executed concurrently. To handle that, some specific mechanisms were used by Gpu-qin. Because this problem is very specific to the GPGPU domain, it will not be discussed in this document.

5.3.1 Fault injection methodology

Although a physical injection technique allows to represent faults in a more realistic way, a software fault injection was chosen for the Gpu-qin project. Indeed, the cost associated with the physical approaches are very high while a software fault injection is low-cost and gives more controllability and repeatability. However, the limited coverage and representativeness are a disadvantage.

To inject faults, *cuda-gdb* has been used. *cuda-gbd* is a debbuger provided with CUDA. It allows to set breakpoints on specific instructions. Once a breakpoint is hit, a fault injection can be performed. It implies that the injection is done at assembly level. An approach based on hardware simulation was also envisaged to inject faults at a lower level. However, the Gpu-qin developers preferred the first solution because of its simplicity and scalability.

Only single bit-flip faults in the functional units of the GPU processor were considered. Although Gpu-qin is able to support multiple bit-flips, this kind of faults was not tested in the study detailed in [18]. The goal was to model transient faults. Permanent faults were not considered because they generally are caused by manufacturing problems while transient ones are induced by external events.

The methodology used by the Gpu-qin tool can be cut out in four phases. The two firsts are very typical to the GPGPU applications and focus on threads. The first step consists of grouping threads according to their behavior. Each group is classified according to its "popularity". One thread from each group is then chosen randomly to be profiled in the second phase. During the second step, the chosen threads are profiled. An execution trace is obtained for each of these threads. That allows to map high-level instructions from the source code to assembly instructions that will effectively be executed. The third phase is the fault injection. To perform it, one of the trace obtained during the second phase is chosen randomly. However, the popularity of the group of threads is taken into account, meaning that a trace from a popular group has

more chances to be selected than one from a less popular group. An instruction from the trace is chosen randomly. A breakpoint is set on this instruction. Once this breakpoint is hit, the program stops its execution. The fault is then injected through cuda-gdb before resuming the normal execution. Finally, the fourth step simply consists of the aggregation of the results.

5.4 Existing fault injection tools

We will here describe existing tools designed to inject faults into a system. Most of them are academic or commercial. For each of these tools, we will give a short description and their pros and cons.

5.4.1 Xception

Xception is a monitoring tool and fault injector that relies on the debugging and monitoring features (e.g. breakpoints registers) available in most processors in order to inject faults. The injector is built like an exception handler. The tester can set breakpoints on specific actions, corresponding to hardware exceptions. For example, Xception can use the "data access" hardware exception. When a particular memory address is accessed (set through the breakpoint), the exception is raised (like it was a memory area that the program is forbidden to reach). Then the injector catch the exception, perform an injection, while the tested program continues running like nothing happened. If the address is read by the application again, the same actions will be performed [19] [8].

The data flow errors can easily be modeled. However, to simulate a control flow error, a breakpoint should be put on the program counter register. It is then difficult to inject a fault that will allow to jump to a desired instruction.

Advantages	Drawbacks
• No source code modification. • No use of software trap.	• Need to be adapted for each (type of) processor. • Hard to model control-flow errors. • Not automated.

5.4.2 DOCTOR

DOCTOR, which states for integrate**D** s**O**ftware fault inje**CTi**n envi**R**onment, is a faults injector that can simulate transient, permanent and intermittent faults into the memory, the processor registers and even the network (which makes it particularly suited to test real-time systems). It uses three different techniques of fault injection : time-out, traps and code modifications. This tool allows the tester to inject single or double bit-flip faults, corrupt an entire byte or alter several bytes at the same time in the memory. The location of these injections can be either chosen by the user or selected randomly by the injector. Control-flow errors can be simulated by injecting faults into the CPU [20] [8] [21].

Also, note that DOCTOR is intended to be flexible and highly portable. To do so, a modular design, which, among other, allows separation of system-dependent parts has been used. Furthermore, a high-level fault injection approach has been chosen [20].

The data flow errors can easily be modeled. However, to simulate a control flow error, the program counter register should be altered. It is then difficult to inject a fault that will allow to jump to a desired instruction.

Advantages	Drawbacks
• Portable and flexible. • Good controllability. • Injection can be automated.	• Hard to model control-flow errors.

5.4.3 FTAPE

FTAPE is a tool designed to test the dependability of a system. FTAPE is divided in three distinct components that can easily interact. The first component (and the most interesting one in this case) is the fault injector, which is implemented as a SoftWare Implemented Fault Injection (SWIFI) tool that can inject faults into the CPU, memory and I/O components. The second part of FTAPE is its workload generator, that is used to simulate program running on the system by soliciting CPU, memory and I/O components. The last component is the measurements unit that can measure the workload on each part of the system. With these three parts, we can see that the tool is intended to be used autonomously to test the entire system. However, it is possible to use each component independently [22] [8] .

The faults injector of FTAPE can inject the following types of errors [22] :

- CPU : bit-flips and set faults in the following registers : floating point registers, program counter, global pointer and stack pointer. Only these registers are considered because they will more likely lead to faults propagation. An injection on the program counter can then allow to inject a CFE. However, this approach makes it difficult to control it.
- Memory : bit-flips and set faults are injected into the memory. The most used parts of the local or global memory are targeted.
- I/O : fault are injected to simulate problems in the SCSI or disk errors.

Advantages	Drawbacks
• Can simulate I/O errors. • Injection can be automated.	• Hard to model control-flow errors. • Few controllability. • Designed to test an entire system instead of programs.

5.5 Comparison

In this section, the criteria that we want of our solution will be put in perspective compared to the test approaches described above. Also the major advantages and drawbacks of the proposed injection tool will be defined. The main limitations of each technique are summarized in the table 5.1.

TIMA	MARS	Gpu-qin	Xception	DOCTOR	Ftape
• No CFE	• No CFE	• No CFE	• Hard to model CFEs	• Hard to model CFEs	• Hard to model CFEs
• Special hardware required	• Not automated	• Only one type of fault considered	• Not automated		• Few controllability
• Random	• Random	• Random	• Not generic		• Few controllability
• Manual analyze of the results					

Table 5.1: Main limitations of the presented fault injection approaches

The principal weakness found in the approaches detailed previously is that none of these solutions allows to model clearly and easily control-flow errors. The test framework implemented for this thesis should be able to simulate each kind of CFE by jumping from an instruction to another.

A main drawback found in several of the solutions presented above is that the fault injections are performed randomly. This can be at a random time, at a random location or both. Such a strategy lacks of controllability, repeatability and reproductibily which are three major characteristics when it comes to test the robustness of a program [7]. The tool developed in the context of this thesis should allow to select the instructions on which faults are injected. The approach should not be random at all but depends of the considered instruction. Also, the tester should have the possibility to choose which bit(s) will be flipped.

Another disadvantage is the use of specific hardware to be able to use the proposed test methodology [17] [14]. That can be either some hardware used to perform physical injections or to facilitate software injections. Of course, use of such techniques will increase the cost. Our solution must perform fault injections through software only.

Finally, another weakness that can be found in several of the solutions presented above is that they are not automated [19] [16]. Indeed, they require the tester to write injection test scripts. In our case, all the tests scripts should be generated automatically. After generation, the tester should only have to run a general script that will launch each injection script and report the results for all the tests.

Chapter 6

Problem statement

This chapter will detail the problem tackled in this thesis. We will first give more information about the context and the constraints it implies. Then the chosen solution will be conceptually presented along with the reasons of this choice.

6.1 Context

To recall, the goal of the thesis is to find and implement a way to automatically test the robustness of a program. This solution should be at the software-level and no special hardware should be required to perform the tests.

The tool developed to perform these tests should be the most generic possible and should test the robustness of any source code. Naturally, it is not really relevant to perform such tests on non-robust programs. Also, if a specific hardener has been used to make the tested program robust, this approach will indirectly allow to verify the validity of the hardening rules used.

The fault injection tests are the last step in a testing process. Indeed, for it to work, a set of unit tests are required. The coverability of these tests are supposed to be maximum. They also – in addition with the functional tests – prove the correctness of the program under test and, if a hardener has been used, prove the equivalence between a program and its hardened counterpart. Based on these unit tests tests, the fault injection tests are used to check the robustness capabilities of the program and, indirectly, to challenge the hardening rules. The entire testing process of a hardened program is represented on figure 6.1. The scope of the problem tackled in this thesis is colored in green.

6.1.1 Errors

We will consider three characteristics to classify the errors that our framework needs to manage : the logical error location, the error type and the physical error location. The entire set of supported errors is presented in table 6.1.

The fault type used to model data flow errors will be bit flips. The number of bits to flip and their position will be defined by the test strategy used.

For the program path, only control flow errors need to be simulated. Other types of wrong execution of an instruction is already represented by data flow errors. Thus, permanent errors on the program path will not be simulated because such control flow errors could often lead to an infinite execution (e.g. a jump from an instruction to a previous one).

The two system-level error models retained in chapter 2 are taken into account in our set of errors. However, only one of the two hardware-level error models is considered, the single bit-flip. As we considered bit flips on several bits, the single bit-flip is a subset of the data flow errors we want to simulate. Contrariwise, the single stuck-at was not considered for two reasons. First, it could lead to irrelevant tests. For example, take a single stuck-at error on a variable that forces a bit to one. If this particular bit is already set to one and never changes through the program execution, the test will have no effect. Second, we considered that the bit flips performed through transient and permanent errors on the data flow were sufficiently representative as a the single stuck-at error is finally a special case of permanent error on a

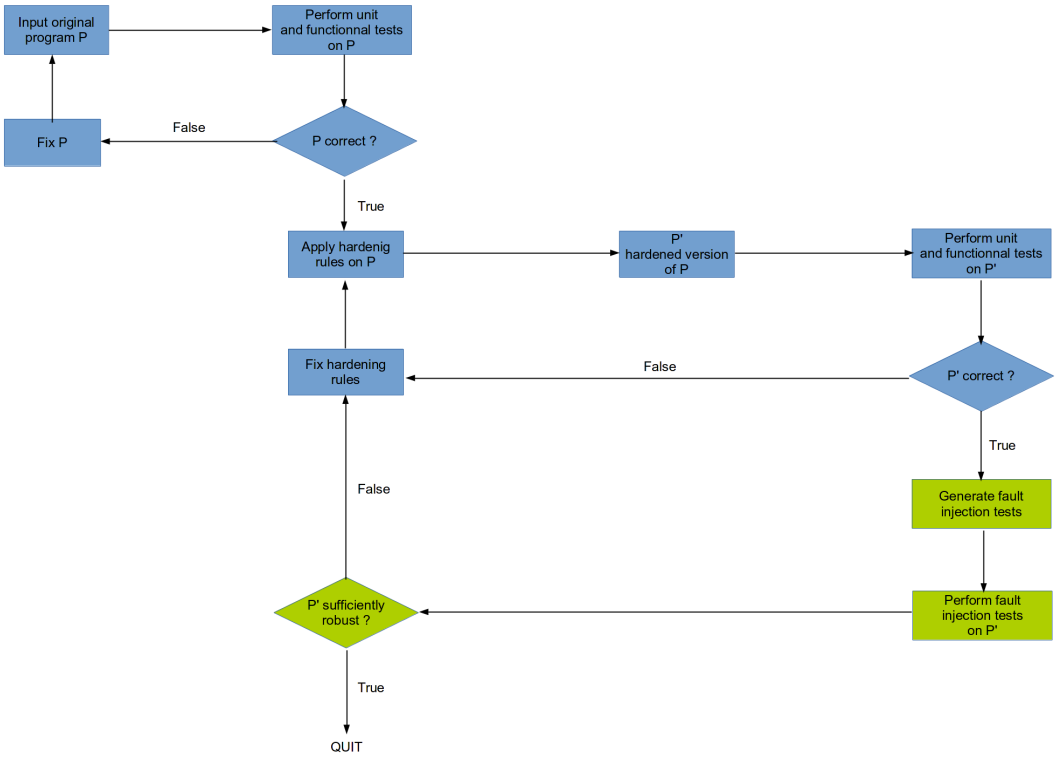


Figure 6.1: testing process

variable where a specific bit could never be modified. Furthermore, all kinds of control flow errors are taken into account through the errors on program path.

6.1.2 Constraints

There are two major constraints related to the framework developed in the context of this thesis.

First, as mentioned previously, the testing tool should be the most generic possible. That implies that it should support a maximum of the C language construct. Moreover, it should be totally independent of any hardening rules or error detection mechanisms. Finally, it should be able to simulated each kind of faults, on each type of instruction.

Second, the testing strategies used with the framework for generating the tests should be the most customizable possible. That implies that the tool should be modular enough to allow the tester to easily write and use its own testing strategy without knowing anything about the core of the framework, i.e. the tester just need to write his strategy definition and it should work without any modification to the framework. Also, the tester should be able to use different strategies in parallel as he found them complementary.

6.1.3 Test cases

To test the framework developed in the context of this thesis, simple hardening rules have been used [2], as presented previously. Note that the hardening capabilities implemented though these rules are not supposed to be the best ones, but to represent the general hardening principles.

Furthermore, a specific context to this project is to use the framework on source code examples provided through the CETIC's hardener. As this is a project lead by the research center,

Logical error location	Error type	Physical error location	
		Storage	Processing
Data path	Transient	A piece of data is incorrect	The output of a single operation might be wrong.
	Permanent	A memory cell fails.	A hardware instruction produce erroneous results.
Program path	Transient	The code of executing software is modified.	A single execution of an instruction not behave as expected

Table 6.1: set of errors

samples of source code produced by this tool have been used to test the framework. This hardener takes a C source code as input and gives another C source code as output. A constraint of the input code is that some features of the C language cannot be used because the hardener only support a subset of the language. The output source code will be functionally equivalent but will add some mechanisms that will allow to detect some types of errors. Note that this hardener was used as a black-box, as the hardening mechanisms it implements were not known as the project is confidential. Thereby, we supposed that the code samples provided were representatives of the hardening rules used. Furthermore, due to this confidentiality aspect, no code sample will be presented in this document. Only the results given by experiments on these programs will be discussed.

The CETIC's sample of code allowed us to have another concrete case of validation. Moreover, it was possible to test the tool on different types of robust source code, proving its generic nature.

6.2 Main approach

We will here describe the chosen technique without going into full details and the reasons of this choice. The primary concepts on which our solution will rely will also be discussed.

6.2.1 Choice

As presented in the background chapters, several solutions are available when it comes to test hardened software. Therefore, to make a choice, the different possibilities should be envisaged and their advantages and drawback discussed.

First, all the solutions implying special hardware equipments were discarded. Indeed, even if these solutions are the most efficient and allows to make tests the closest to the reality, the drawbacks are numerous. First, the injections rely on randomness as specific faults cannot be simulated. Second, the costs required to acquired or built such equipment are very high while these techniques does not have a lot of advantages. Moreover, such an approach was not relevant in this context because only a software solution could be envisaged. Similar conclusions can be drawn for the solutions implying simulation. In this case, the cost can be measured in time because it is really complex to implement a good simulator or an appropriate framework to allow to corrupt the emulated system. Furthermore, the reachability and controllability of

such methods is quite low. Moreover, a solution implying simulation would rely on randomness because it is difficult to make the link between the simulated hardware and the variables or instruction of a program executed on it. Injecting specific control flow error would have been almost impossible.

Second, the possibility of mutation testing was envisaged. Indeed this solution has a lot of advantages and does not rely on randomness as the physical solutions. However, the original software needs to be modified. This is a big drawback because the tested software is not the same. The test results could then be affected by the applied mutation. Furthermore, this approach implies that a lot of mutants programs are created. This results in a lot of almost exact copies of the same codeS.

Finally, the software fault injection during the execution also seems to be an interesting possibility. Indeed it has the same advantages as the mutation testing approach but the original source code is not directly modified. Moreover, contrary to mutation testing, a lot of mutants are not created. However, the main drawback of this kind of method is that it can leads to a lot of useless test cases.

In conclusion, the choice needed to be made between fault injection at compilation-time or at run-time because of the advantages of these two solutions, mainly the controllability which is important when specific faults need to be simulated. Ultimately, the second solution was chosen. The fact that the original source code was not modified was a great advantage in the context of this thesis. Furthermore, injecting faults during the execution of the program under test is closer to a real situation than simply modifying the source code.

6.2.2 Concept

Objectives

The goal is to simulate faults during the execution of the program. To achieve so, we want to identify for each instruction which faults could be generated on it. It will allow us to inject specific faults when the program reach this specific statement.

To recall, the primary objective behind these fault injections is to verify the robustness of the program under test. The secondary goal is to challenge the hardening rules used to make this program robust. Indeed, if the program cannot resists to some kind of errors, it could reflect a lacuna in these rules.

Tests

The result of a fault injection experiment is based on a unit test. However, the results interpretation differs from the tests discussed in chapter 2 as they are only used to validate the implementation of the hardening rules. For these tests, we can be sure that no error happened. Contrariwise, in the case of a fault injection we know that a fault occurred. The goal is then to verify the good operation of the error detection mechanisms. Based on the example from chapter 2, the possible results along with their interpretation can be found on table 6.2.

As we can see on table 6.2 the test pass only if the error flag is different from zero, which means that the error detection mechanisms have been triggered. The last case is clearly a failed test because the fault had an effect on the value returned but was not detected. The first is more nuanced. We cannot mark it as a success because the error was not actually detected. The fact that the result was not affected could be a coincidence. Moreover, the fact that the fault is not detected can reflect a weakness of the hardening rules. However, it is possible that the injected fault had no effect on the execution. A manual inspection is required. This special case and the consequences it brings will be discussed in more details in the validation chapter.

Returned value	Error flag	Test result	Observation
TRUE	0	Failed	No error detection mechanism have been triggered although the result value is good.
FALSE	!= 0	Passed	The result value is not the one expected and the error detection mechanisms have been triggered
TRUE	!= 0	Passed	The result value is the one expected but the error detection mechanisms have been triggered.
FALSE	0	Failed	The result is not the one expected and no error detection mechanisms has been triggered.

Table 6.2: results table for fault injection tests

Strategies

Another particularity of our approach is the customization possibilities brought by the use of strategies. A strategy is a way of defining which (type of) tests are generated by configuring parameters. It is the strategy itself that manage the test generation. Some examples are given on table 6.3.

Strategy	Faults category	parameters	Description
Percentage transient variables	Data flow	pvar : percentage of variables bit : bit to flip	Produce fault injection scripts that allows to inject a transient single bit flip fault on the selected bit for pvar% of all the variables contained in the tested functions.
Partial permanent variables	Data flow	nvar : number of variables	Produce fault injection scripts that allows to inject a permanent fault on only nvar number of all the variables contained in the tested functions.
Percentage blocks CFE 1	Control flow	pblocks : percentage of basic blocks	Produce fault injection scripts that allows to simulate a CFE of type 1 between only pblock% of all the basic blocks.
Partial statements CFE 5	Control flow	nstmts : number of statements	Produce fault injection scripts that allows to simulate a CFE of type 5 between only nstmts statements of each considered basic block.

Table 6.3: examples of strategies

As we can see, the strategies can take variables parameters as they require. Naturally, one can combine several strategies if he found them complementary.

Chapter 7

Proposed solution

We will describe here the solution implemented in full details. To recall, a run-time software fault injection technique was chosen. We will first describe the methodology used to achieve the run-time fault injection. Second, the entire test process using our proposed solution will be detailed along with the functional architecture of the developed tool. Third, the technical architecture will be discussed. This will allow us to describe the main components used for the tests generation. Finally, each component will be explained and more details will be given about the features and technical challenges met during the development of the solution.

7.1 Methodology

To allow to inject faults at run-time, the GNU Debugger (GDB) will be used. Thanks to this tool, the program can be stopped at any instruction during its execution via a *breakpoint*. The debugger also allows to modify the content of a program variable. A GDB script will then be written for each fault(s) injection test. For each fault injected through a specific script, a breakpoint will be set on the instruction on which the fault should be injected. A set of commands will then be associated to this breakpoint in order to simulated the desired fault.

As we said earlier, we do not want our tool to rely only on randomness but it should allow to simulate defined errors instead. To achieve that, specific errors should be injected for a given instruction. It is then necessary to analyze each instruction and decide which fault(s) need to be simulated on it (if any). To allow us to go through the entire code instruction by instruction we decided to use a *Abstract Syntax Tree* (AST). This AST will be produced by a parser and will allow us to analyze each line of code independently. By going through this AST, the instructions containing variables on which faults could be injected will be identified. A GDB script defining a breakpoint that can be set on the instruction will be written and specific corruption commands will be associated to this breakpoint.

The AST approach is relevant in order to know which data flow faults need to be injected depending upon the instruction. However, this is not convenient for errors on the program path (CFEs). As we have seen, these errors relies on the control flow of the program. A good representation of that is the Control Flow Graph. So, A Control Flow Graph will be build for each function. Then, based on the identified basic blocks, scripts to inject different kinds of Control Flow Errors will be produced.

Finally, once all the GDB scripts have been written, some kind of run scripts will need to be generated too. These scripts will allow the tester to run all the fault injections on the target program. Once an injection has been executed, the result of the test will be written in a result file.

7.2 Functional architecture

The developed solution can be divided into six main steps, a presented on figure 7.1.

First, the representations (AST and CFG) need to be generated because the test generation will be based on them. Then, the properties to configure the strategies and other global

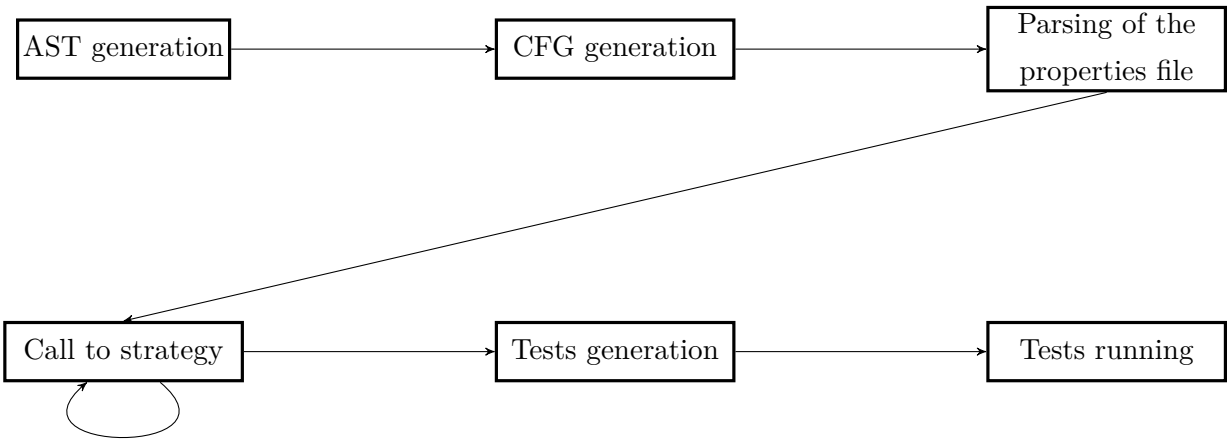


Figure 7.1: Compile-time fault injection process

parameters are loaded. Then each chosen strategy is independently launched and the tests are generated. The last step is the execution of these tests.

7.3 Technical architecture

We will detail here the organization of each component composing the architecture of our tests generation tool. The components diagram ca be found on figure 7.2.

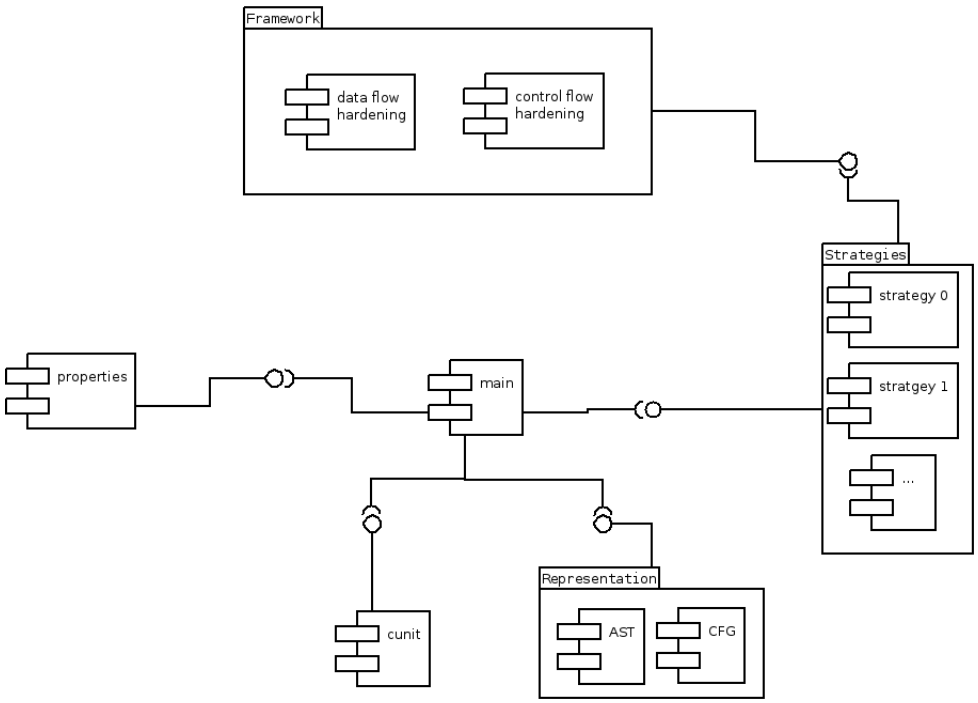


Figure 7.2: UML components diagram

As it appears on the figure 7.2, the main component makes the link between the other parts. First, it manages the generation of the representation used (an Abstract Syntax Tree and a Control Flow Graph). Then, the main component manages all the arguments that need to be given to the program and to each specific strategy. It loads them from the properties component in order to launch the strategies. Once the strategies are run they call the components composing

the framework to generate the adequate tests. After the generation, the main component also generates all scripts necessary to run the generated faults injection scripts. Finally, if the cunit support is required, the main component call the cunit component to transform the tests.

The sequence diagram show in figure 7.3 show the tests generation process.

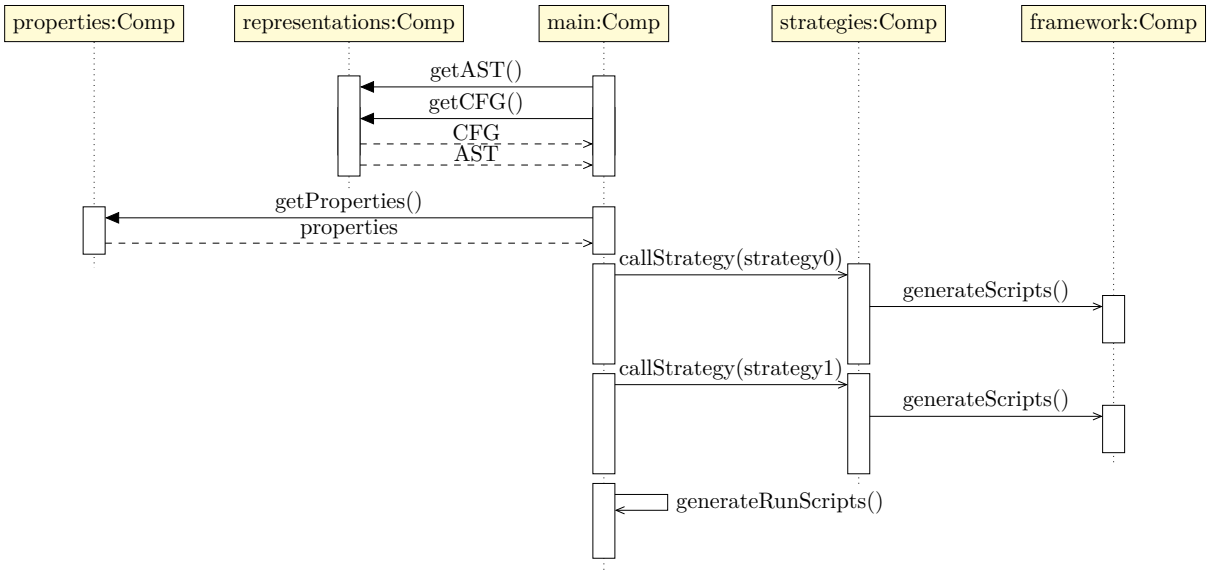


Figure 7.3: sequence diagram of the generation of tests scripts

Based on this architecture, the following sections will detail each part of the solution. These sections are more technical as details and challenges that were met will be discussed.

7.4 Framework

The framework is the part of the solution that allows to generate the tests scripts. The functions it defines can be easily called by a user defining a custom strategy. The main separation in the framework part lies in the differentiation between the generation of data flow and control flow errors.

7.4.1 Data flow

As seen previously, the errors affecting the data flow can be separated in two ways. Indeed, such faults can be transient or permanent and they can either affect the data storage or the data processing. The part of the framework managing the data flow errors allow to generate scripts for these four kinds of faults. However, before generating these faults, the user defining its test strategy must have access to the statements on which the faults can be injected. To do so, the data flow part of the framework provide a function that return a dictionary of all relevant statements (or parts of statements) on which one or more fault(s) can be applied. The statements are separated into three groups : the statements affected by operators, the "left" statements that represents the directly modified statements (e.g. left part of an assignation) and the "right" statements which are the statements used to compute another variable or expression but are not modified directly by the statements (e.g. right part of an assignation). For each category, the statements are grouped in lists regarding the operators or variables they affect.

Based on all these statements, a strategy can generate the desired scripts. We will then describe the generation of these scripts. To illustrate the explanations, examples of such scripts will be given based on the "running example". The source code of this example program is given below :

example.c

```
1  void procedure(int *res0, int *res1) {
2          ecf = 0;
3
4          int a0, a1;
5          int b0, b1;
6
7          b0 = 0;
8          b1 = 0;
9          a0 = b0 + 1;
10         a1 = b1 + 1;
11
12         check(b0 == b1);
13         check(efc == 0)
14
15         if (a0) {
16                 ecf = 1;
17                 check(a0 == a1);
18                 check(a0);
19
20                 b0 = 1;
21                 b1 = 1;
22
23                 *res0 = a0;
24                 *res1 = a1;
25
26                 check(a0 == a1);
27                 check(ecf == 1);
28         }
29         else {
30                 ecf = 2;
31                 check(a0 == a1);
32                 check(!a0);
33
34                 *res0 = b0;
35                 *res1 = b1;
36
37                 check(b0 == b1);
38                 check(ecf == 2);
39         }
40 }
```

Transient data storage faults

A transient data storage fault is an error that corrupt one instance of one variable in the entire program.

This kind of fault can be injected on any variable. Once the desired instance is found, a breakpoint is set on the line containing this instance. Once the breakpoint has been hit, a command will corrupt the variable as specified by the used strategy (number of bits to flip, which bits, etc.). A differentiation can also be made between the injections *a priori* and *a posteriori*. In the first case, the injection is performed before executing the line on which the breakpoint have been set, in the later it is performed just after. Also, the breakpoint needs to be temporary as a transient fault can only happens once.

Here is an example based on the running example code :

GDB script

```
1  break example.c:8
2          commands
3                  corrupt b1 1 -1 next disable
4          end
5
6  run
```

As you can see, the GDB script given here will break on the line 8 of the example code. Once it is done, the `corrupt` command will be called to inject a fault on the `b1` variable. The `corrupt` command is a custom GDB command that allows to flip bit(s) in order to inject a fault

on a given variable and resume the program execution after the injection has been performed. It works as follow :

```
corrupt variable_name numbers_of_bits_to_flip bits_to_flip [next] [disable]
```

where :

- `variable_name` is the name of the variable on which the fault will be injected.
- `number_of_bits_to_flip` represents the number of bits that will be flipped to corrupt the variable.
- `bits_to_flip` is a list of all the specific bits that will be flipped. -1 means that the bit is chosen randomly.
- `next` is an optional argument. If it is given, the injection will be made *a posteriori*, if not it will be performed *a priori*.
- `disable` is an optional argument. If it is given, the breakpoint will be disabled after the first time it is hit.

So, in this example, a fault will be injected on the `b1` variable *a posteriori* on one bit chosen at random. Also, as it is a transient fault, the breakpoint is temporary.

Transient data processing

A transient data processing fault is an error that corrupt one instance of one variable that has been affected by an operator in the entire program.

The approach is almost the same as for the transient data storage faults, only the type of statements affected will change. Here is an example :

GDB script

```
1  break example.c:9
2          commands
3                 corrupt a0 1 -1 next disable
4          end
5
6  run
```

As for the case of transient data storage error, one random bit of the corrupted variable is flipped *a posteriori* and the breakpoint is temporary.

Permanent data storage

A permanent data storage fault is an error that corrupt one variable each time a "write" operation is performed to update its value.

This kind of fault can be injected on any variable. Every time a statement corresponding to a write operation performed on this variable is found, a breakpoint is set on the line containing the write instruction. Every time one of the breakpoints has been hit, a command will corrupt the variable as specified by the used strategy. To simulate this kind of fault, only the injection *a posteriori* should be taken into account. Indeed, because the variable is corrupted after each update operation, its value will already be corrupted when it is used in another context. However, the choice is up to the tester who writes his strategy. Finally, the breakpoint cannot be temporary as it is a permanent fault.

Here is an example :

GDB script

```
1  break example.c:8
2          commands
3                 corrupt b1 1 -1 next
4          end
5
6  break example.c:21
7          commands
8                 corrupt b1 1 -1 next
9          end
10
11 run
```

In this example, a fault is injected on the `b1` variable *a posteriori* each time one of the two write instructions applied on it are executed.

Permanent data processing

A permanent data processing fault is an error that corrupt one variable each time a write operation using one specific operator is performed to update this variable.

The approach is very similar to the one used for permanent data storage faults, only the statements are chosen differently. Here is an example :

GDB script

```
1  break example.c:9
2          commands
3                 corrupt a0 1 -1 next
4          end
5
6  break example.c:10
7          commands
8                 corrupt a1 1 -1 next
9          end
10
11 run
```

In this example, the variables affected by the plus operator will be corrupted each time an instruction representing a write operation using this operator will be executed. As for the previous example, the injection affect one random bit and is performed *a posteriori*.

7.4.2 Control flow

The control flow part of the framework is composed by functions that allows one to generate GDB scripts that simulate the five different types of Control Flow Errors. Based on the Control Flow Graph, the basic blocks are extracted and jump operations can be performed between the blocks, the blocks' statements or both. Also, some checks are made in order to reduce the number of "false positive" tests. These tests are injections in which the normal control flow is followed.

We will describe here the methodology followed to generate GDB scripts for each kind of CFE.

Control Flow Errors – Type 1

This type of error happens when the control flow takes an illegal branch of the Control Flow Graph. To recall, an illegal branch is one that does not appears in the control flow graph.

Here is an example. See the red line on figure 7.4. As it appears, there is normally no path from BB1 to BB0.

```
1  break example.c:27
2          commands
```

```
3               MakeJump example.c:2 next
4       end
5   run
```

So, as you can see, the jump is made from the end of BB1 to the beginning of BB0. The `makeJump` command is a custom GDB command that allows to jump to another instruction. The command can be used as follow :

```
1   MakeJump file:line [next]
```

The first argument is the coordinate to which the jump will be performed. The second is optional and specify if the instruction on which the breakpoint has been set should be executed before making the jump operation or not. In this specific case, the last instruction of the block needs to be executed before injecting the error. Also note that all the breakpoints used to inject Control Flow Errors are temporary because generating permanent CFEs often results in an infinite loop as the jump can lead to a previous instruction.

Control Flow Errors – Type 2

This type of error happens when the control flow takes a wrong branch of the control flow graph. To recall, a wrong branch is one that exists in the control flow graph but should not be taken during the execution.

An example is the purple line on figure 7.4. As you can see, the normal execution path should take the "if" branch (from BB0 to BB1). If the taken branch is the one from BB0 to BB2, a wrong branch is used.

```
1   break example.c:15
2       commands
3         if (a0)
4               MakeJump example.c:31 next
5             else
6       MakeJump example.c:16 next
7         end
8   run
```

Note the condition that appears in the command block of the breakpoint. Indeed, as BB0 is a conditional block, the condition of its last instruction is used in order to reduce the number of false positive tests. So, this test is relevant for any execution of the program, no matter the path taken by the normal control flow.

In the case of a basic block having several possible output branches but no defined condition, one test per branch will be generated. A false positive result will appears and the tester will need to manually check this specific test case.

Control Flow Errors – Type 3

A CFE of type 3 occurss when a jump is performed at the end of a basic block that leads to a random instruction of another basic block.

An example is the green line on figure 7.4. The associated GDB script is given below.

```
1   break example.c:15
2       commands
3             MakeJump example.c:22 next
4       end
5   run
```

Control Flow Errors – Type 4

A type 4 CFE happens when a jump is performed from a random instruction of a basic block that leads to a random instruction of another basic block.

An example is the blue line on figure 7.4. The associated GDB script is given below.

```
break example.c:15
        commands
                MakeJump example.c:22 next
        end
run
```

Control Flow Errors – Type 5

A type 5 CFE is intra-block and happens when a jump is performed from a random instruction of a basic block that leads to a random instruction of the same basic block.

An example is the cyan line on figure 7.4. The associated GDB script is given below.

```
break example.c:9
        commands
                MakeJump example.c:2 next
        end
run
```

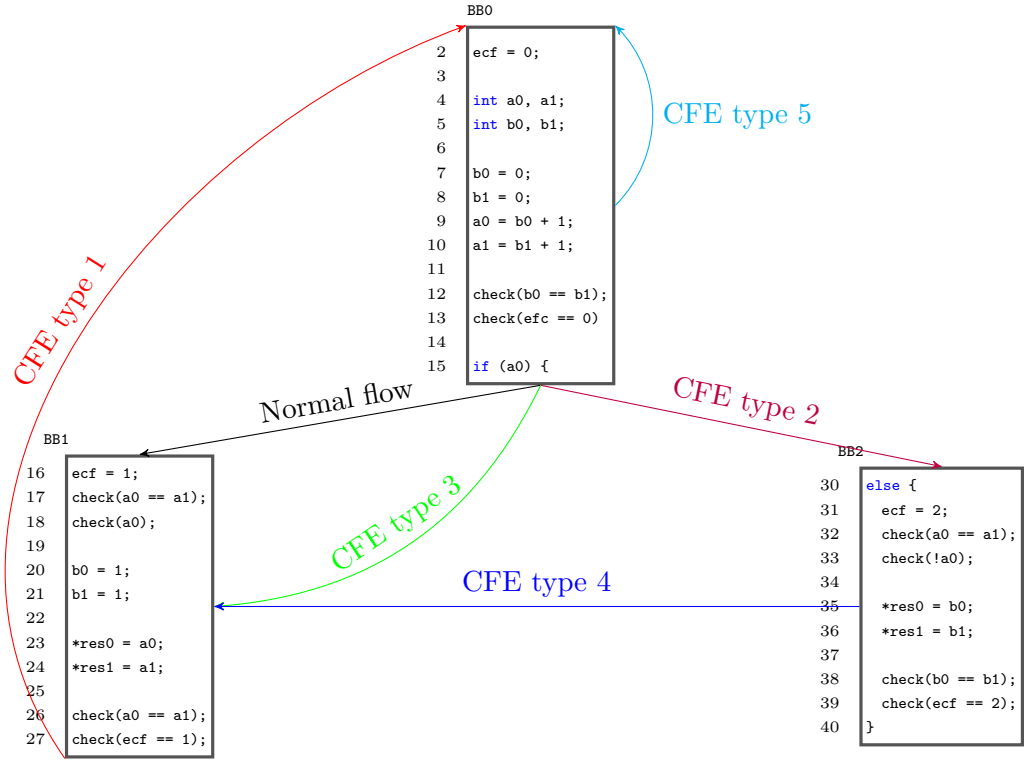


Figure 7.4: Control Flow Graph – example code

7.5 Strategies

As explained, the framework can be used to generate several different kinds of GDB test scripts. The way these tests are generated, the parameterization of the generation and the number or types of tests that are produced are defined by a *strategy*.

7.5.1 Define a strategy

As the tool is modular, it is easy for a tester to create his own strategy. A strategy is no more than a python function written into a python file placed in a particular folder. Once one wants

to generate the tests following one strategy or combining several of them, he just have to run the main python script and give it the name of the python files in which the strategies he wants to use are defined.

7.5.2 parameters

Of course, one can give as many parameters he wants to his custom strategies. To give all correct arguments to them, a properties file under the json format is available. The list of custom arguments each strategy needs must be defined in this file. Also, general arguments need to be given to the program through this file. Note that some parameters are mandatory and then common to all strategies. The mandatory arguments and the name of the function executing the strategy are the only two imposed conventions. Besides that, the tester has a total freedom in the way he will generate his tests.

7.6 Generation

A main file, named `run.py` following the common convention, manage all the generation. It has several roles.

First, it generates the necessary representations to be used by the strategies to generate the tests. These representations are the Control Flow Graph, the statements dictionary used by the data flow part of the framework and the Abstract Syntax Tree, necessary to create the statements dictionary.

Second, it creates all the directories based on the corresponding argument given in the properties file.

Third, based on other parameters, it creates a Scons files. A Scons file has approximately the same use as a makefile and will be used to compile the necessary files in order to run the tests.

Fourth, it initialize the run scripts specific to each kind of faults. These run scripts will be completed when the tests are generated and launched by the main run script when one wants to run the tests.

Finally, it goes through the Abstract Syntax Tree, extract the functions for which GDB script will be written and creates the statements dictionary needed by the strategies. Then, for every one of these functions, it runs the selected strategies.

7.7 Cunit

At the request of the CETIC, an interfacing with the popular unit testing framework Cunit has been implemented. This allows one to give a Cunit test file to the program. This file should contains unit tests for a non-hardened program. Then, this file will be transformed into an equivalent Cunit file but adapted to the hardened version of the program. Also, other modifications are made to the GDB test scripts in order to be used with the unit testing framework.

The Cunit support is quite limited as only one type of Cunit assertion can be used. Furthermore, this part of the tool is not very generic. The general Cunit support can be used with any Cunit file if it only makes use of supported features and add some special instructions to be compatible with the fault injection approach. However, the automatic transformation of the Cunit file to its "hardened-compatible" counterpart can only be used with code hardened through the CETIC's tool. As each hardening technique and implementation has its own specificities, it is difficult to transform these kinds of unit tests to be adapted to all hardening approaches.

As an example, a simple Cunit test file and its hardened-adapted counterpart will be presented. Note that the "normal" version is intended to work with the non hardened version of

the example procedure as it is presented in the chapter 2 while the second version is adapted to the hardened version of the same procedure. Remember that this transformation must be made manually as the support for the automatic adaptation is provided for the CETIC's source code only.

Normal version

```
1  #include <CUnit/CUnit.h>
2  #include "CUnit/Basic.h"
3  #include "CUnit/Console.h"
4
5  int init_suite(void) { return 0; }
6  int clean_suite(void) { return 0; }
7
8  void test_example(void) {
9    int res;
10   procedure(&res);
11   CU_ASSERT_EQUAL(res, 1);
12 }
13
14 int main(int argc, char **argv) {
15
16   CU_initialize_registry();
17
18   CU_pSuite pSuite = CU_add_suite( "example_test_suite",
           init_suite, clean_suite );
19   CU_add_test(pSuite, "test_example", test_example);
20
21   CU_basic_set_mode(CU_BRM_VERBOSE);
22   CU_basic_run_tests();
23
24   CU_cleanup_registry();
25   return 1;
26 }
```

Hardened adapted version

```
1  #include <CUnit/CUnit.h>
2  #include "CUnit/Basic.h"
3  #include "CUnit/Console.h"
4  #include "hardening.h"
5
6  #define MY_CU_ASSERT_EQUAL(exp,result) \
7    test_result = !error_flag; \
8    CU_ASSERT_EQUAL(exp, result); \
9    fi_init();
10
11 int test_result = 0;
12
13 void fi_init(void) {
14   error_flag = 0;
15 }
16
17 int init_suite(void) { return 0; }
18 int clean_suite(void) { return 0; }
19
20 void test_example(void) {
21   if(!fork()) {
22     int res0, res1;
23     procedure(&res0, &res1);
24     MY_CU_ASSERT_EQUAL(res0, 1);
25   }
26 }
27
28 int main(int argc, char **argv) {
29
30   CU_initialize_registry();
31
32   CU_pSuite pSuite = CU_add_suite( "example_test_suite",
         init_suite, clean_suite );
33   CU_add_test(pSuite, "test_example", test_example);
34
35   CU_basic_set_mode(CU_BRM_VERBOSE);
36   CU_basic_run_tests();
37
38   CU_cleanup_registry();
39   return 1;
40 }
```

As you can see, the `CU_ASSERT_EQUAL` is replaced by a custom version `MY_CU_ASSERT_EQUAL`. This allow to check the test results via the `test_result` variable. This will be useful to conclude on the fault injection results as it will be discussed later in the adequate section. Also, the error flag should be reinitialized between each assertion. This is done thanks to the `fi_init` function.

Another difference is that each test is executed independently in a specific process. This is needed because some fault injections can cause the interruption of the program because of a signal (e.g. segmentation fault). If this happens, only the child process in which the test is executed will be killed instead of the main program.

7.8 Tests scripts

This section will explain how the GDB test scripts are launched once they have been generated. The general organization and operation will also be explained.

7.8.1 Organization

To contain and organize the tests scripts, some folders are created. The root directory contains the general run script and the Scons file. The root directory also contains one sub-directory for each function on which fault injections will be performed. In each of these sub-folders, there is a run script to launch test for a particular function. There are also three sub-directories : one for transient faults on the data flow, one for permanent faults on the data flow and one for Control Flow Errors. These directories contains sub-directories to organize the tests scripts. For the data flow folders, there are two sub-directories that differentiates between tests on data storage and processing. For the Control flow directories, there are five sub-directories, one for each type of CFE. These sub-folders are the lasts of the tree and holds the GDB tests scripts and a run script that allows to run all the tests contained in the directory.

7.8.2 Run the tests

To run the tests, one just need to run the main run script from the root directory. This script will compile the program through the Scons script, and launch the run scripts specific to each function folders. These scripts will then run the ones specific to each kind of faults for the functions the folder they are in represents.

After running the tests, the root directory will contains the result files and the compiled version of the program to test. The functions sub-folder will also contain a result file. Finally, the sub-directories containing the GDB scripts will contain one result file per test.

7.8.3 Results

The global and function specific results files contains the following information :

- Test name
- How much breakpoints have been hit
- The unit test result
- The fault injection result

The results file are in XML format. However, a more user-friendly HTML version of the global results file is also available. An example for each entry can be found on table 7.1.

Test	Breakpoint	Test result	FI Result
`CFE3_1-to-2_59`	1	1	FI detected
`transientDataProcessing_37_a1_bits--1_0`	1	0	Failed
`transientDataStorage_57_b1_bits--1_0`	0	0	No injection
`transientDataStorage_5_res0_bits--1_0`	1	0	IGSEG

Table 7.1: HTML result file

As you can see, there are four different results type for the fault injection results :

- Green means that the test passed, i.e. an injection has been performed and detected.
- Red means that the test failed, i.e. an injection has been performed but not detected.
- Orange means that no injection has been performed.

- Yellow is the default color if anything else happens. In general, as in this example, a signal has interrupted the execution.

Listeners

To collect the results of tests, specific listeners have been added to GDB. An "exit listener" is triggered when the test finish its execution, check the relevant information and write in the results file. Also, a "signal listener" has been implemented in order to catch signal that stop the program. When it is triggered, it also writes into the result file.

If the Cunit option is used, this approach is not relevant anymore. Indeed, the fault injection result should be caught between each cunit assertion. To acheive so, a special breakpoint is put on the `fi_init` function. When this breakpoint is hit, the test value is checked thanks to the `test_result` variable. The result of the fault injection can then be drawn. Also, this breakpoint allows GDB to reinitialize some values or options in order to be ready for the next test.

7.9 Conclusion

To summarize, our solution implements a run-time fault injection technique through the GDB debugger. Specific fault injection scripts are generated following defined strategies through the framework. These scripts allows to put breakpoints on relevant instructions and custom GDB commands are triggered went they are hit. These commands allows to effectively inject the fault, by flipping bits of a variable or provoking a jump in order to simulate a control flow error. With this approach, all the errors from or error model can be simulated.

These scripts must be used with a unit test. Once all the injections have been performed, the results are reported in specific files.

Chapter 8

Validation

To validate both the tool and the test methodology used, some experiments will be carried out.

First some benchmarks will be performed. This will allow us to see the evolution of the number of tests generated through the framework. To achieve so, we chose to configure the strategies such as the maximum number of tests are generated for each type of fault. Some source code samples from trivial programs were used. This allowed us to compare the effect of three factors on the number of tests generated : the number of lines, the number of variables and the number of basic blocks.

Second, the framework will first be challenged by being used on two different kinds of robust code. Then, the strategies allowing to reduce the number of tests will be evaluated. The results of the experiments will be presented and discussed.

Finally, some limitations that have been identified through these experiments and measures will be discussed.

Note that we combine the errors on data processing and data storage when the results of our experiments are reported. These are actually the same errors as the data processing errors are simulated by altering the value of the variable they affect. In the results, these errors are reported as "transient" and "permanent".

8.1 Benchmarks

The number of tests have an influence on the generation and execution time. The different measures presented below will allow us to draw some conclusions related to the factors that influence the number of generated tests. For these experiments, the maximum numbers of scripts of each kind of error have been generated. The results are reported in table 8.1 and displayed graphically on figure 8.1.

8.1.1 Measures

Factors			**Number of test scripts**							
size(lines)	variables	basic blocks	transient	permanent	CFE 1	CFE 2	CFE 3	CFE 4	CFE 5	Total
3	3	1	4	7	0	0	3	9	5	28
5	3	1	9	4	0	0	5	25	15	58
7	2	3	3	2	4	1	12	14	4	40
11	3	5	15	4	13	5	45	74	18	174
12	3	6	14	5	35	5	63	70	9	201
30	6	11	33	8	116	9	252	415	35	868
40	11	12	48	13	136	13	390	876	74	1550

Table 8.1: Number of tests generated

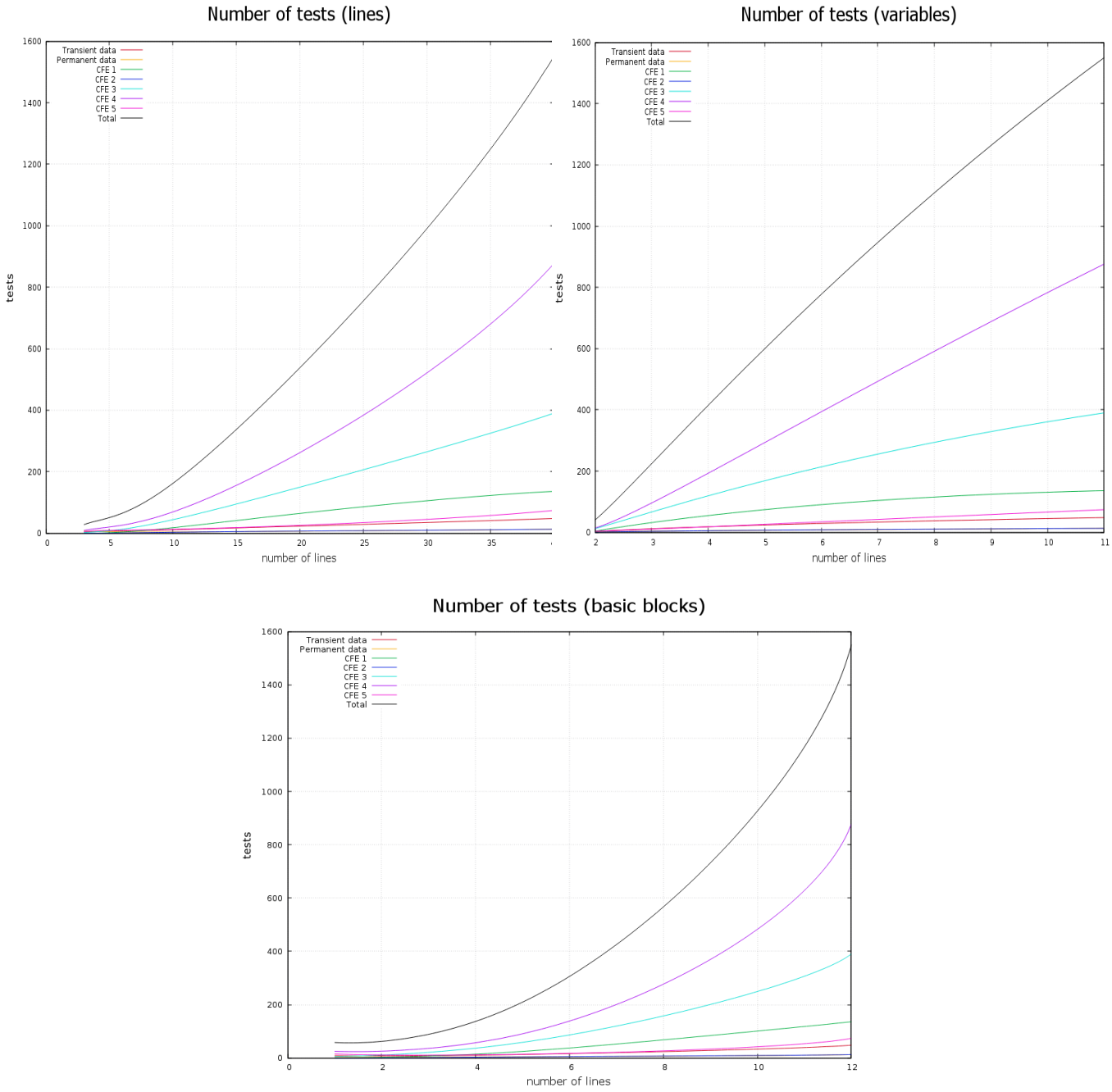


Figure 8.1: Number of tests generated

8.1.2 Analyze

As the graphics show, the factor that affect the most the number of tests generated is the number of basic blocks. Indeed, as we can see, it strongly affects the number of scripts simulating control flow errors, especially for the type 3 and 4 while the number of data flow tests grows linearly and slowly. That implies that the total number of tests generated increases exponentially depending upon the number of basic blocks.

The two other factors, the number of lines and variables, have no sensible effect on the number of generated tests as it increases in a almost linear manner depending upon these factors. However, the size of the source code have a bit more effect than the number of variables because it logically increases the probability of having more basic blocks.

Of course, the generation and execution times are affected in the same way because they directly depends on the number of tests.

8.2 Experiments

8.2.1 Fault injections

In order to test the framework, a simple program was hardened. From the original program whose source code is given below, two versions were created. The first was hardened manually through the hardening rules described in [2]. The second version was created through the CETIC's hardener. The test framework was used on these two versions. Fault injection scripts were generated for the maximum of all kinds of the set of supported errors. These tests were run with the unit test given next to the program source code (adapted depending upon the hardening technique). These experiments allowed to prove the effect of the errors simulated by the scripts. Moreover, as the two versions make use of totally different hardening rules, it proves the genericity of the framework.

```c
#include"exponent.h"

int times(int a, int b) {
        if(a == 0) {
                return 0;
        }
        else if(b == 0) {
                return 0;
        }
        else {
                int tmp = a;
                while(b > 1) {
                        tmp = tmp + a;
                        b = b-1;
                }
                return tmp;
        }
}

int f_exponent(int element, int exponent) {
        if(exponent == 0) {
                return 1;
        }
        else {
                int tmp = element;
                while(exponent > 1) {
                        tmp = times(tmp, element);
                        exponent = exponent - 1;
                }
                return tmp;
        }
}
```

```c
#include"exponent.h"

int main(char** args) {
        int res;
        res = f_exponent(4, 2);

        return res != 16;
}
```

Note that the tested code is very simple and is tested with only one unit test. The first reason is that the results of the injections performed on a simple example are easier to analyze. The second is that, as we have seen previously with our hardening example, the transformations brought to the source code through the hardening process implies that new lines are added. For the CETIC's hardening rules, the scaling factor is pretty high. As we have previously experimented, more lines means more tests generated. This implies a longer execution time to perform these tests. Note that, for the CETIC's version this simple code, more than thirty hours have been needed to perform the injections on the `f_exponent` procedure only.

To perform our experiment, the unit test was also modified for the hardened versions. We decided to check the value of the error flag only. Indeed, the goal is to verify that the error detection mechanisms have been triggered after the fault has been injected.

The results of the experiment on the two robust versions of the code are reported in the table 8.2.

Source code	Result types	Injected error types							
		Transient	**Permanent**	**CFE1**	**CFE2**	**CFE3**	**CFE4**	**CFE5**	**Total**
Example	Detected	74	10	7	2	139	1390	99	1721
	Failed	38	1	4	2	65	660	162	932
	No injection	19	0	4	0	51	510	52	636
	Signal	1	2	0	0	0	0	0	3
	Total	132	13	15	4	255	2560	313	3292
CETIC	Detected	105	30	23	3	964	34331	4610	40066
	Failed	496	54	7	3	34	1970	270	2834
	No injection	27	0	7	0	197	756	49	1036
	Signal	8	3	12	0	381	1656	15	2075
	Total	636	87	49	6	1576	38713	4944	46011

Table 8.2: Tests results

To analyze these results, we will use the graphic shown on figures 8.2 and 8.3 (a) and (b). The figure 8.2 show the proportions for each test result for each kind of fault. The two others show the same information for the whole test set.

At first sight of the graphics, the CETIC hardening rules seem to be a lot more effective than the ones from [2] if we base our analyze on the figures 8.3 (a) and (b) representing the global results. However, the details given by the figure 8.2 shows that the tests on data flow performed on the CETIC version have failed in a lot more cases than the ones performed on the other program. Te proportion of failed tests seems enormous. After analyzing them a bit more, it appears that a lot of failed tests are marked as "failed" because the error flag has not been modified though the returned value is correct. The figure 8.4 shows the same information than the 8.2 one. However, a new "warning" category has been added to mark the proportions of these tests marked as "failed" but where the return value is the one expected. As the problem occur mainly in the case of data flow errors, only the results for these kinds of error is reported on figure 8.4. As we suspected, the big part of failed tests is composed of tests where the returned value is correct even if the error detection mechanisms were not triggered.

The interpretation of such a result is tricky. Indeed, in the case of a classic hardening test, the test could be considered as it passed, as mentioned in chapter 2. However, in this case, we know that an error happened and no error detection mechanism have been triggered, even if the result value is correct. That sort of "false positive" result could happen, for example, if the error is injected on a data that is not used after the injection or not used at all. Also, some special kind of computation could make that the error have no effect on the final result. Consider for example the instruction below :

```
1    //error injected on a
2    result = a + 58 - a;
```

As you can see, an error injected on the `a` variable will not have any effect on the result value. On the other hand, even if the result value is correct, the error has not been detected and this could reflect a problem in the hardening rules. Therefore, this case should not be classified as a test that was successful.

For the control flow errors, we can see that the CETIC's approach is clearly more efficient than the other one. Except for the type 2, the CETIC version of the program detects a greater proportion of errors and fails in a much lower number of cases. As presented in chapter 2, the approach described in [2] has some limitations for the control flow hardening. Therefore, these results are not surprising.

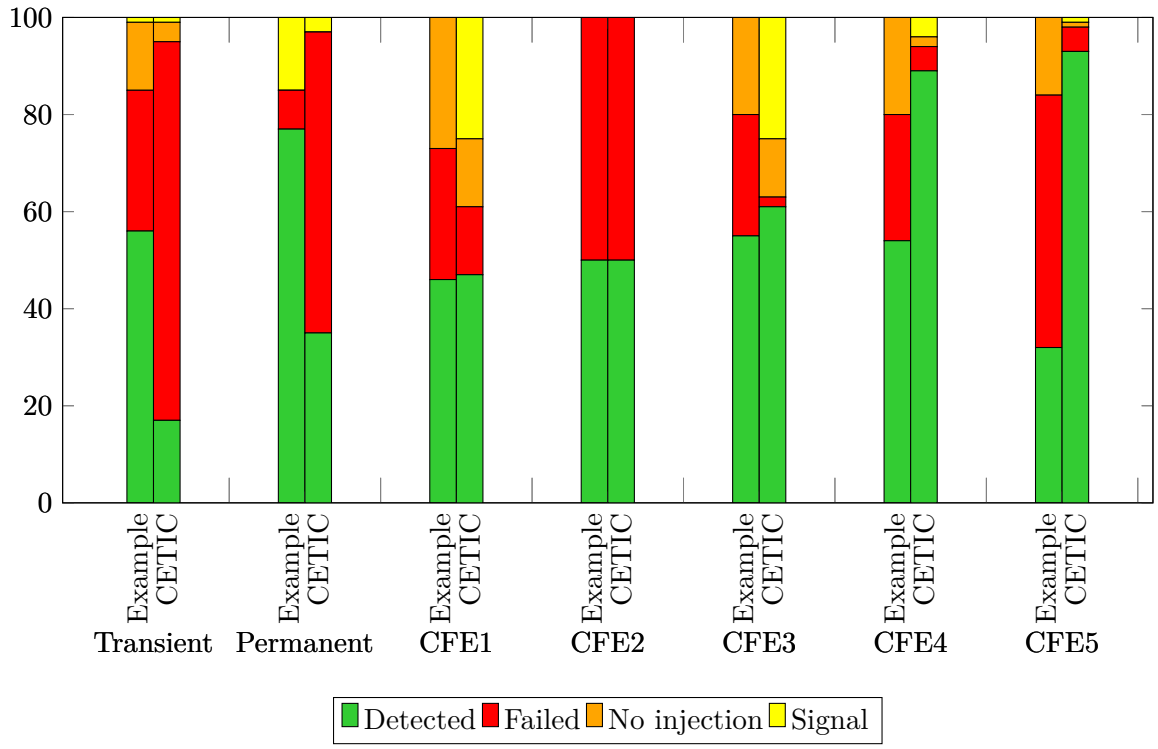
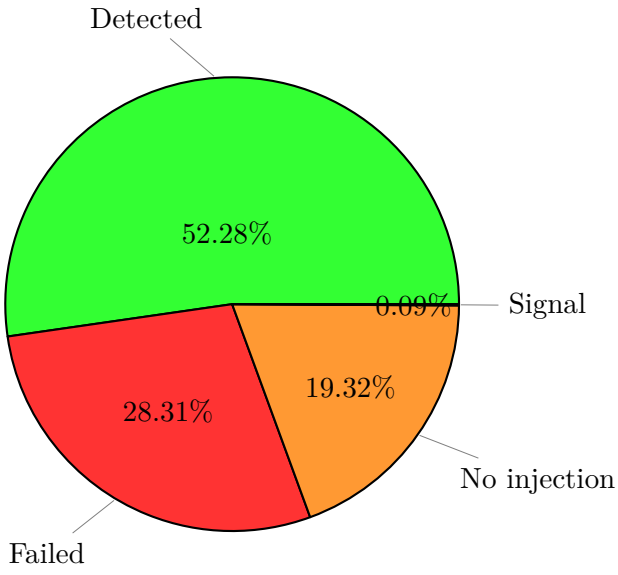
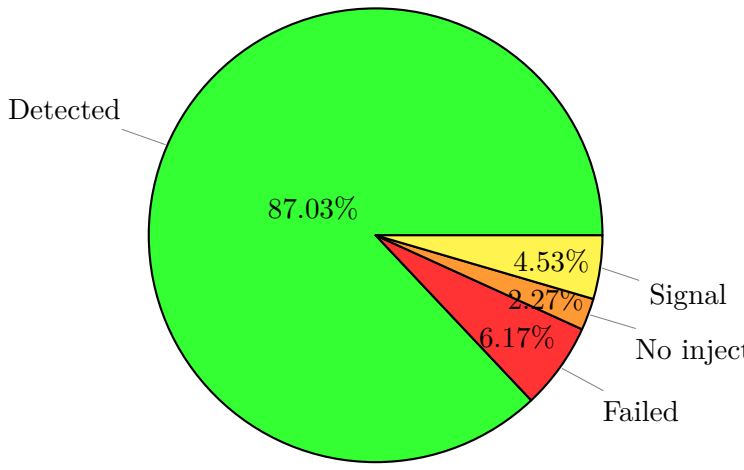


Figure 8.2: Graphical results of the experiment



(a) Example version



(b) CETIC version

Figure 8.3: Results of fault injections

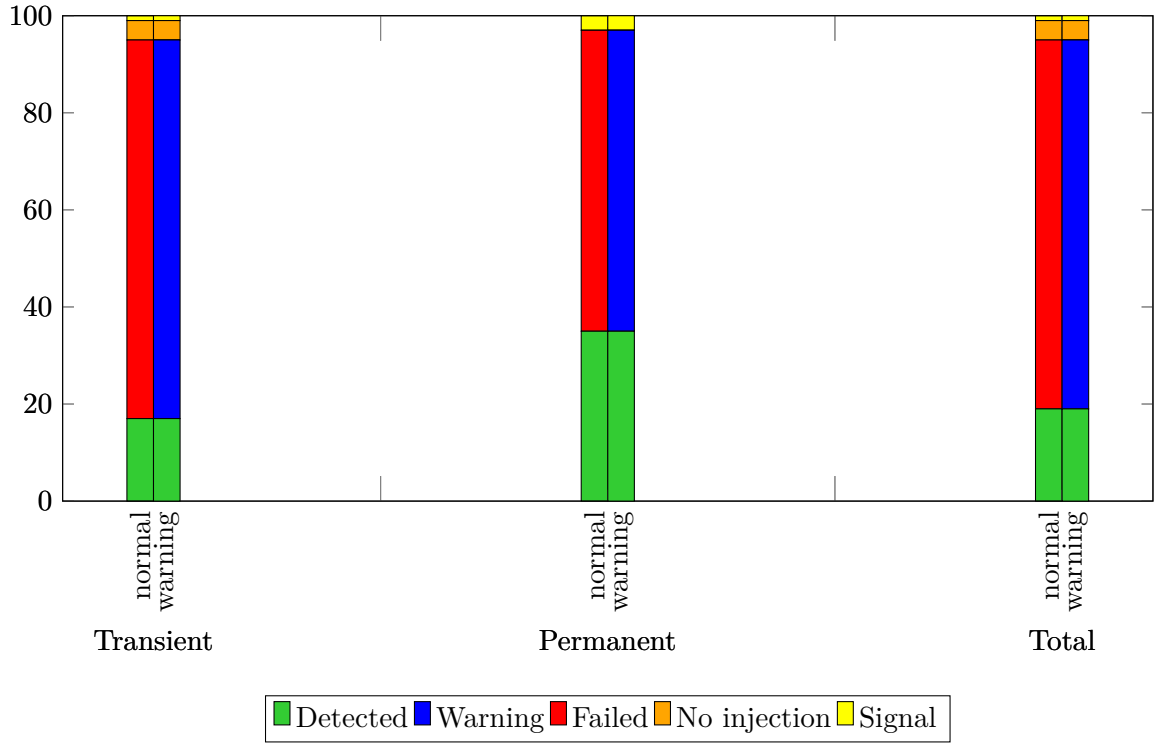


Figure 8.4: Graphical results of the experiment with "warning" category

8.2.2 Strategies

For the previous experiments, we configured the strategies to generated the most possible tests. However, these strategies can be configured differently. To perform this second experiment, four factors will be used to parameterize the tests generation :

- The percentage of variables : that will be taken in account for the generation of data flow tests.
- The percentage of statements : that will be taken in account for each variable considered for data flow tests.
- The percentage of basic blocks : that will be taken in account for control flow tests.
- The percentage of lines inside each basic blocks : that will be taken in account for control flow tests.

Of course, this will allow to reduce the number of tests generated. This could be a solution to the enormous number of tests which are generated. However, we cannot be sure that the efficiency of the test will be the same if some cases are not tested. We tried to measure the differences obtained in the results if we reduce these percentage to 50% and 25%.

The results are reported on the table 8.3 and graphics shown on figures 8.5 and 8.6.

As the figures show, the results seem to stay consistent as we lower the configurations to reduce the number of tests for global results. However, if we look at each kind of error independently, we can see that some error types seems to be more affected than others. Thus, the results for the control flow errors seems to be strongly affected while the ones for data flow errors seems to stay consistent. For a concrete example, have a look on the CFE of type 4 for the example approach. Furthermore, the 25% strategy seems to generate a too small sample of test to have relevant results.

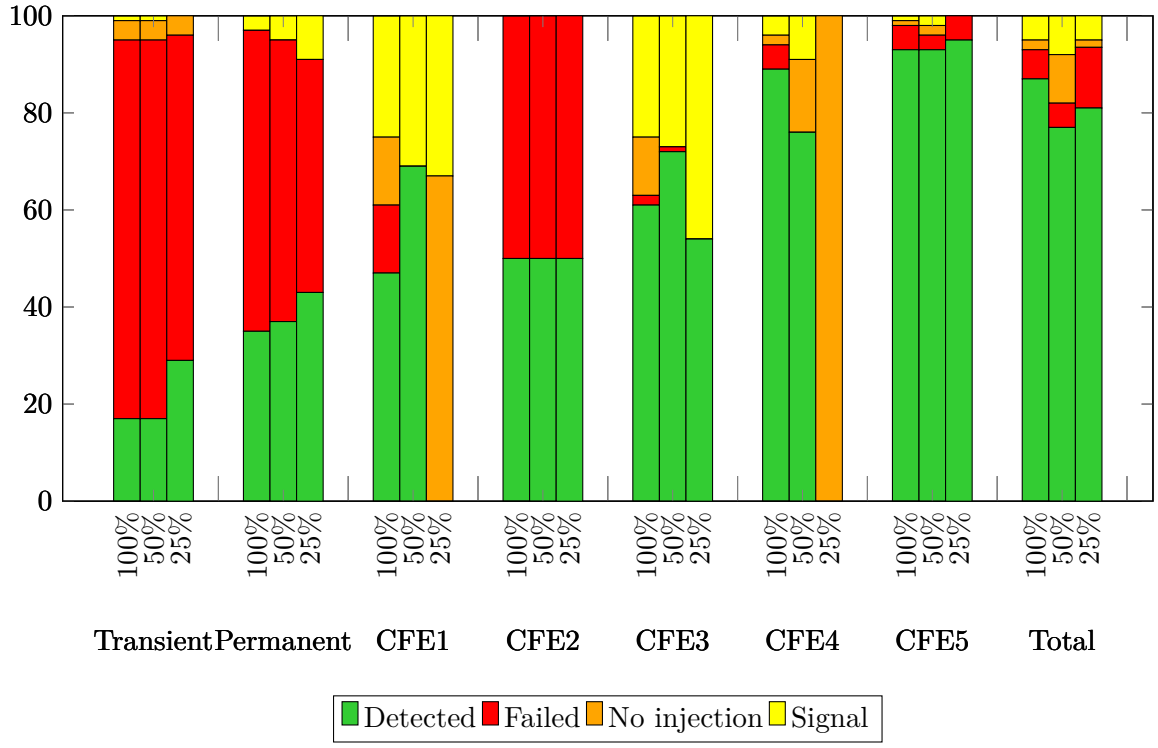


Figure 8.5: Graphical results of the strategy configuration experiment for the CETIC approach

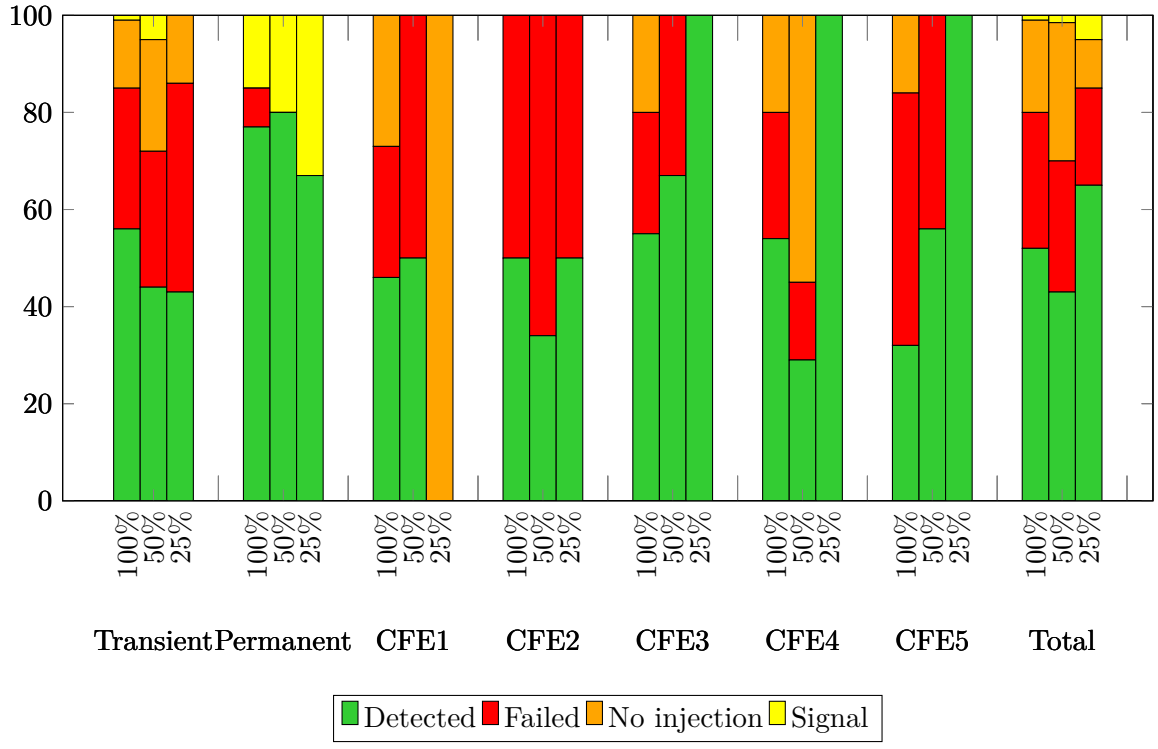


Figure 8.6: Graphical results of the strategy configuration experiment for the example approach

Source code	Percentage	Result	Error types						
			Transient	**Permanent**	**CFE1**	**CFE2**	**CFE3**	**CFE4**	**CFE5**
CETIC	100%	Detected	105	30	23	3	964	34331	4610
		Failed	496	54	7	3	34	1970	270
		No injection	27	0	7	0	197	756	49
		Signal	8	3	12	0	381	1656	15
	50%	Detected	20	16	9	2	172	1609	727
		Failed	93	25	0	2	3	4	25
		No injection	5	0	0	0	0	325	18
		Signal	2	2	4	0	65	193	9
	25%	Detected	7	9	0	1	15	0	209
		Failed	16	10	0	1	0	0	11
		No injection	1	0	2	0	0	2	0
		Signal	0	2	1	0	13	0	16
Example	100%	Detected	75	10	7	2	139	1390	99
		Failed	38	1	4	2	65	660	192
		No injection	19	0	4	0	51	510	52
		Signal	1	2	0	0	0	0	0
	50%	Detected	17	4	2	1	12	26	27
		Failed	11	0	2	2	6	14	21
		No injection	9	0	0	0	0	50	0
		Signal	2	1	0	0	0	0	0
	25%	Detected	3	2	0	1	2	4	1
		Failed	3	0	0	1	0	0	0
		No injection	1	0	1	0	0	0	0
		Signal	0	1	0	0	0	0	0

Table 8.3: Results with different strategies

Note that the tests reduction strategies seems to have less effect on the CETIC version of the code. Our guess is that the CETIC version having a lot more lines of code and variables, there are still a relevant number of tests with the 50% – and even the 25% for some types of error – to make the fault injections set representative. With the other version, as we can see from the table 8.3, there are a very small number of test cases generated, mostly when the 25% configuration is used. For example, we can see that only 4 tests are generated for the CFEs of type 4 for the 25% configuration while there are 2560 for the 100% configuration.

In conclusion, these configurations options can be used to reduce the number of tests. However, the reduction possibilities depends on the number of variables for data flow configurations and of basic blocks for control flow ones. The reduction proposed here (or any other) should only be used on big source code, in a reasonable way.

8.3 Limitations

We will here detail the limitations that we could identify in both the test approach and the framework.

First, the benchmarks and experiments have shown us that the number of generated tests can rapidly explode on a code of significant size. Indeed, the testing phase can then take up to several days on a code of a few hundreds or thousands of lines if all the tests need to be performed. However, to limit this effect, strategies can be implemented and/or configured to reduce the number of tests. Only a portions of the variables and/or basic blocks could be taken

into account. Nevertheless, less tests means a greater chance to miss an error or an interesting "borderline case" as we demonstrated. A more interesting approach to reduce the number of tests could be to spot the equivalent fault injections in order to only perform one of them. For example, consider a simple addition operation `a + 1`. This part of statement will provoke two equivalents tests : the data storage one and the data processing one. Indeed, the `a` variable will be corrupted in these two cases.

Second, a problem was identified in the interpretation of some specific tests. To recall, when an injection has been preformed and the error flag has not been modified whereas the result value is correct, the test is marked as "failed". The reasons are explained in the previous section. However, it feels wrong to mark this test as "Failed". A special result indicating a warning should be used to mark these special cases. However, as the result of the test is determined by the return value of the unit test, it is no so easy to adapt the tool to achieve that.

Third, a limitation of our framework is the fact that only one type of fault can be simulated at a time. It could be interesting to combine some injection. We considered that, in a majority of cases, it would not be relevant. Indeed, if one of two combined fault injected is detected and not the other, the "failed" case would not be reported as the test will be marked as a success. However, the possibility could be attractive to define new test approaches. For example, to reduce the execution time, it could be an improvement to combine tests of faults that must be injected in two different basic blocks such as the execution of one implies that the other is not (e.g. the "if" and "else" branch in a CFG).

Fourth, another limit is that the fault injections are performed per function as each function is taken independently to generate the tests. Moreover, no "inter-function" tests can be performed via this approach. The case of control flow errors is easily defensible as a jump from one function to another would mostly lead to a segmentation fault. Moreover, the number of scripts to simulate control flow errors would be strongly higher. However, for permanent data flow errors, it could be interesting to inject the fault through all function calls. For example, such an error injected on a variable in a given function will only be altered in the scope of this function. If another function is called and the faulty variable is used as a parameter it should be corrupted each time it is used in the called function.

Fifth, some restrictions can be found at the injection location level. Indeed, the errors are injected from a high-level and directly on the variable. It could be interesting to use a "lower-level" approach in some specific cases. For example, permanent errors should be injected on a specific memory location through the entire program execution and not on the "high-level" variable.

Finally, a limitation can be found at the hardware location level as errors are simulated at the memory-level only. No errors are injected at the processor-level (e.g. through the registers) or at the transfer-level. Note that it is not possible to alter information at transfer-level with the software approach used in this context.

8.4 Conclusion

The experiments lead with the tests generated through the fault injection framework developed allowed us to identify some weaknesses from the hardening approaches used as test cases. Also, the condition of genericity was respected as the tests on different types of robust program have proved. The goal that was to test a robust program and challenge the hardening rules used to produce it was achieved.

Via these tests, the test approach was also challenged. As the code and hardening mechanisms of the version of the exponent program hardened manually through the rules from [2] were known, we could say that the test approach used challenged every hardening rules. As we supposed this example representative of any hardening mechanism, we can suppose the coverability of our approach is adequate.

Moreover, we managed to make our test approach and tool architecture compatible with the use of custom strategies that could be defined by a user. This complete another one of our original objectives.

However, as detailed in the previous section, some limitations in both our tool and testing approach have been found. Moreover, some improvements could be made. These are discussed in the next chapter.

Chapter 9

Future work

After the tests were performed with the tool and the analysis of the results, we spotted some limitations. We will discuss here the aspects that can be enhanced on both the developed tool and the test methodology used.

9.1 False positive result

Sometimes, the result file show that an injection has been performed but the error has not been detected by the program under test. However the corruption did not influence the execution and the test result is correct. The fact that the fault was not detected is then not important.

For example, imagine a variable that is corrupted after it has been used for the last time. After its value has been read, it is corrupted. The held value, although it is incorrect, will never be used again. The injection is then irrelevant. It is an efficiency issue.

Some of these cases could be avoided. During the tests generation, a check could be made in the abstract syntax tree to verify that the variable on which the error will be injected will be used after injection. If it is not the case, the test should not be generated.

Another possibility is that "dead code" is present in the tested function. For example, consider a variable that is declared then initialized but never used. This variable is totally useless and could be removed from the source code. However, such a variable exists in the program, at least one injection fault script will be written for it by the test framework. Of course, even if this variable is corrupted, it will have no effect to the program execution. We considered that the dead code issue could be handled by the programmer before performing any test.

9.2 Execution path

The test strategy does not take the execution path into account. It implies that some tests will be run when the fault injection need to be performed on a line that is never reached by the program. The breakpoint will never be hit and the injection will not be performed.

Imagine a function with a if-then-else instruction. Naturally, for a given unit test, only one path will be followed depending upon the condition : either the "if" or the "else". However, fault injection scripts will be written for the two branches. As a result, even if the "if" branch is taken, all the script related to the "else" branch will be tested.

This is also a efficiency issue. Furthermore, because of the large amount of injection scripts, this can have a serious impact on performance as this can make the execution time a lot longer.

To avoid this situation, a symbolic execution technique as described in [23] could be used prior to execution of fault injection tests. Based on the unit test, the execution path could be evaluated. Thenceforward, only the tests that take place on the execution path should be launched.

9.3 Results

As presented in the solution chapter, the results files generated when the tests have finished is pretty basic. A nice to have feature could be to add several more information in it. A few examples could be the following. For the data flow errors, the bits corrupted could be added. For the control flow errors, the lines from which the jump has been made and to which it leads could be a useful information. A more complex improvement could be to report the line on which the error detection mechanism has been triggered.

9.4 Running options

Some parameters could be added to the scripts used to run the tests. For example, the tester could choose which kind of tests he wants to launch and/or on which functions.

9.4.1 Time-out

Sometimes, an injection can make an execution (almost) infinite. For example, imagine a permanent fault injected on the `counter` variable in the code below :

```
1  while(counter != 0) {
2    counter--;
3  }
```

if the tester is unlucky, it can take a very long time before the injection give the value zero to the `counter` variable. To avoid this kind of problem, a time-out should be available for the tester to define. It could be one parameter to give to the run script. If one test lasts more than the time-out value, it should be killed. Note that this functionnality exists if the Cunit support feature is used.

9.5 Threads

As we could see in the validation chapter, the time needed to perform the tests can be very high for source code of significant size. Therefore, it could take up to hours or even days just to run all the tests on a code of a few hundreds of lines. To reduce this testing time, the use of several threads could be an improvement. For example, we could continue to test function by function but make a separate thread for each kind of faults or make one thread per function.

9.6 Unit testing frameworks

As previously stated, an option exists to transform unit tests written using the Cunit framework for a non-hardened code in order to make them compatible with the CETIC's hardened code. Also, other transformations are applied to make them usable with the tests framework developed in the context of this thesis. An improvement could be to make this feature more generic and compatible with other hardening rules. Moreover, only a few Cunit features supported. For example, only one type of assertion can be used. Support for each type of assertion could already be an improvement. Finally, support for other popular testing framework could be a nice to have feature.

9.7 Errors

9.7.1 Error models

As we discussed previously, the error model is composed of only a restrained set of errors. This could be interresting to extend it. For example, the single stuck-at error model could be added.

9.7.2 Combining errors

We only considered one (type of) error injected per script. In the tests cases used, this choice was relevant because we wanted to check the detection capabilities for each type of errors independently. However, combining injections could be useful in some cases. For example, we could imagine a hardened program that uses different error flags depending upon the error type. In this case, two injections could be performed simultaneously and the detection could be verified for each of them.

9.8 Combining tests approaches

As we saw, the test methodology we chose has some limitations. First, as a software approach, the representativeness of the injected fault is quite low. Second, some hardware locations cannot be reached (e.g. injections at the transfer-level). To overcome these drawbacks, our approach could be combined with another one.

A first idea could be to combine our technique with a hardware-based or simulation-based approach. By doing so, faults could be injected at locations that are inaccessible with our approach. Event if these faults would be randomly injected, these previously inaccessible locations could be tested. Moreover, some other types of error could be modeled like *non-logical errors* (e.g. the bit value becomes something between 0 and 1).Then, for more precise and controllable error injections, our software method could be used.

Another possibility could be to combine our injection method with mutation testing. That could allow to inject faults on mutated programs and then combine fault simulations. This could be a adequate way to, for example, make tests combining processing errors (the operators are mutated) and control flow errors (via run-time software fault injection). As the mutation process is heavy, the mutation testing should be used for permanent errors only. Moreover, the mutation testing approach could add some functionalities to the tested program. For example, to resolve the previous problem of the infinite loop execution, the mutation could implement a second condition to the loops that will prevent this situation to happen.

9.9 Inter-functions injections

By combining abstract syntax trees, the entire program could be represented on one data structure. That could allow to track the use of variables to inject permanent errors through the entire program. That could also allow to produce a more complex control flow graph that will represent more than one function. Based on that, more complex control flow errors could be introduced.

9.10 Conclusion

As we have shown, most improvements can be made on the tool, which could support more features. These features can be very practical, like the enhancement of the results presentation, or more conceptual, like the use of symbolic execution to reduce the number of tests executed.

The test methodology could also be improved. First, a larger spectrum of errors could be considered. Second, different (kind of) errors could be injected in a single test run. Finally, the main limitations to our method are inherent to the run-time software fault injection approach. To overcome these restrictions, the only solution is to combine our technique with other types of approach.

Bibliography

[1] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[2] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," in *Defect and Fault Tolerance in VLSI Systems, 1999. DFT'99. International Symposium on*, pp. 210–218, IEEE, 1999.

[3] P. C. Mehlitz and J. Penix, "Expecting the unexpected-radiation hardened software," *Infocom@ American Inst. of Aeronautics and Astronautics*, 2005.

[4] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pp. 3–9, 1978.

[5] B. Randell and J. Xu, "The evolution of the recovery block concept," *Software Fault Tolerance*, vol. 3, pp. 1–22, 1995.

[6] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-implemented hardware fault tolerance*. Springer, 2006.

[7] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques.," *IEEE Trans. Computers*, vol. 52, no. 9, pp. 1115–1133, 2003.

[8] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[9] S. Rudrakshi, V. Midasala, and S. Bhavanam, "Implementation of fpga based fault injection tool (fito) for testing fault tolerant designs," *IACSIT International Journal of Engineering and Technology*, vol. 4, no. 5, pp. 522–526, 2012.

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

[11] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the C programming language," Tech. Rep. SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN„ March 1989.

[12] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[13] W. E. Wong, *On mutation and data flow*. PhD thesis, Citeseer, 1993.

[14] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," *Nuclear Science, IEEE Transactions on*, vol. 47, pp. 2231–2236, Dec 2000.

[15] A. Datum, J. Reisinger, W. Schwabl, and H. Kopetz, "The real-time operating system of mars," 1989.

[16] E. Fuchs, "An evaluation of the error detection mechanisms in mars using software-implemented fault injection," in *Dependable Computing—EDCC-2*, pp. 73–90, Springer, 1996.

[17] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, and G. Leber, "Integration and comparison of three physical fault injection techniques," in *Predictably Dependable Computing Systems* (B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, eds.), ESPRIT Basic Research Series, pp. 309–327, Springer Berlin Heidelberg, 1995.

[18] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 221–230, IEEE, 2014.

[19] J. Carreira, H. Madeira, J. G. Silva, *et al.*, "Xception: Software fault injection and monitoring in processor functional units," *Dependable Computing and Fault Tolerant Systems*, vol. 10, pp. 245–266, 1998.

[20] S. Han, K. Shin, and H. Rosenberg, "Doctor: an integrated software fault injection environment," in *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pp. 204–213, Apr 1995.

[21] S. Tixeuil, W. Hoarau, and L. Silva, "An overview of existing tools for fault-injection and dependability benchmarking in grids," in *Second CoreGRID Workshop on Grid and Peer to Peer Systems Architecture*, 2006.

[22] T. K. Tsai and R. K. Iyer, "Measuring fault tolerance with the ftape fault injection tool," in *Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems*, MMB '95, (London, UK, UK), pp. 26–40, Springer-Verlag, 1995.

[23] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 367–381, Springer, 2008.