Faculty of Sciences and Technology of the University of Coimbra
Department of Informatics Engineering

Software Verification Aimed at Security Vulnerabilities

Luís Miguel Agante Gonçalo

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering advised by Prof. Raul André Brajczewski Barbosa and Prof. Regina Lúcia de Oliveira Moraes and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra

September 2019

1 2 9 0

UNIVERSIDADE Ð
COIMBRA

This page is intentionally left blank.

Acknowledgements

This master thesis this dissertation represents the climax of an academic journey full of good moments that I will never forget.

I would first like to express my sincere gratitude to my advisors Prof. Raul Barbosa and Prof. Regina Moraes for the support in everything and for steering me in the right the direction.

To my family I must express my gratitude for providing me with unconditional support, continuous encouragement and endless patience throughout my years of study and through the process of researching and writing this dissertation.

To my girlfriend, Jéssica, I am thankful for all the kind words, for all the support and specially for encourage me to continue and not letting me give up.

Last but not least, to all my friends and colleagues thank you very much, this would not be possible without all of you!

This page is intentionally left blank.

Abstract

Software testing is an activity aimed at the identification and later correction of faults capable of reducing the software quality. This activity is essential because the occurrence of mistakes is inevitable and although some of these mistakes are not relevant, others can be expensive and dangerous. Due to this, it is important to test everything that is produced. However, even the best software testing techniques may fail to detect some faults and in many cases these faults result in security vulnerabilities.

This dissertation aims at developing an approach to find the bugs that are prone to create security vulnerabilities and that are usually invisible to the strategies commonly used. In the first phase, the types of mistakes more frequently associated with security vulnerabilities were analyzed. This analysis allows us to realize a representative fault injection to calculate the detection rate of the commonly used testing techniques in order to understand their effectiveness and the type of faults that are more difficult to detect... develop a methodology more focused on these faults. The developed tool is capable of creating test cases based on the constants present in the code. On the other hand, it is also capable of detecting extraneous faults, which are often invisible to the commonly used testing techniques. Afterwards, to verify the feasibility of the developed tool, it was performed a representative vulnerability injection in order to compare the detection rate of the testing techniques studied in comparison with the developed approaches.

Keywords

Testing, Software faults, Fault injection, Security vulnerabilities, Security

This page is intentionally left blank.

Resumo

Software testing é uma atividade que tem como finalidade a identificação e posterior correção de falhas que podem diminuir a qualidade do *software*. Esta atividade é considerada indispensável porque todos cometemos enganos e, apesar de algumas falhas não serem importantes, outras podem ser caras e perigosas. Por esse motivo, é necessário verificar tudo o que é produzido. No entanto, mesmo as melhores técnicas de testes de *software* deixam falhas por detetar e, em muitos casos, essas falhas são responsáveis por criar vulnerabilidades de segurança.

Esta dissertação tem como objetivo desenvolver uma metodologia para a deteção de bugs propensos a criar vulnerabilidades de segurança e que são normalmente invisíveis para as técnicas de teste geralmente usadas. Numa primeira fase, foi feita uma análise aos defeitos mais frequentemente associados a vulnerabilidades de segurança. Esta análise permitiu-nos perceber que tipo de falhas as técnicas de teste falham em detetar e assim desenvolver uma metodologia mais focada neste tipo de falhas. A ferramenta desenvolvida é capaz de criar casos de teste com base nas constantes presentes no código. Por outro lado, também é capaz de detetar código superfluo, que muitas vezes é invisível às técnicas de teste habitualmente utilizadas. Posteriormente, de modo a verificar a viabilidade da ferramenta desenvolvida, foi realizada uma injeção de vulnerabilidades representativas de forma a avaliar a deteção de falhas pelas metodologias de teste estudadas em comparação com as abordagens desenvolvidas.

Palavras-chave

Testes, Falhas de software, Injeção de falhas, Vulnerabilidades de segurança, Segurança

This page is intentionally left blank.

Contents

Acronyms

ACTS Automated Combinatorial Testing for Software.

AST Abstract syntax tree.

DOCTOR Integrated software fault injection environment.

EFC Extraneous function call.

EIFS Extraneous IF construct plus statements.

Ferrari Fault and ERRor Automatic Real-time Injection.

FIAT Fault Injection Based Automated Testing Environment.

FTAPE Fault Tolerance And Performance Evaluator.

G-SWFIT Generic Software Fault Injection Technique.

LFI Library-level Fault Injector.

MCS Minimal cut sets.

MFC Missing function call.

MIFS Missing if construct and surrounded statements.

MLAC Missing AND sub-expr in expression used as branch condition.

MLOC Missing OR sub-expr in expression used as branch condition.

MLPA Missing small and localized part of the algorithm.

MODIFI MODel-Implemented Fault Injection.

OAT Orthogonal Array Testing.

ODC Orthogonal Defect Classification.

PFI Probe/fault injection.

SWIFI Software Implemented Fault Injection.

VM Virtual machine.

WALR Wrong algorithm — code was misplaced.

WLEC Wrong logical expression used as branch condition.

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

Software quality is increasingly becoming essential to all business. Although many factors can affect the quality of the software, for example a careful design or a good architecture, testing is still the primary method used by the industry to evaluate developed software. The concept of software testing can be separated in validation and verification. Validation is the process of evaluating software at the end of the software development process to ensure compliance with the intended usage expressed by the software specification. It normally depends on the knowledge of the application for which the software is written. For example, validation of software for an airplane requires knowledge from aerospace engineers and pilots. Verification is the process of determining if the product fulfills the requirements established in the previous phases. It is usually a more technical activity and requires knowledge about the individual software artifacts, requirements and specifications [14].

Even the most advanced testing techniques may fail. The software is complex and humans are prone to make mistakes, so vulnerabilities tend to occur [15]. As the number of software systems increases, so do the number of vulnerabilities; and considering that most systems are exposed through the internet to multiple users, someone may try to exploit the vulnerabilities to warm the system. New software vulnerabilities are discovered on an almost daily basis and have caused severe damage to organizations [16]. Thus, it is crucial to be able to detect and correct them as early as possible.

The main objective on this dissertation is to develop a methodology that complements the commonly testing techniques on the detection of security vulnerabilities in software written in C programming language. We decided to focus in C as it is widely used for developing system software and system bugs can be more dangerous than application bugs. For example, a vulnerability in OpenSSH can be exploited by an attacker to remotely obtain access to the system and execute unauthorized actions, exposing information from end-users and organizations. Also, according to the TIOBE Index [17], this language has consistently been at the top of popularity for almost 20 years, so it is still a vastly used programming language.

1.1 Motivation

It is well known that, nowadays, software is present in almost everything. It is also known that software possesses a high probability of containing bugs. Some of them do not change the software functionalities but they can be exploited by an attacker to perform unauthorized actions. Apart from the information that can be stolen, there are high costs associated with these attacks. It is projected that *"by 2021 the costs related with cybercrime hit $6 trillion annually"* [18].

One of the most famous exploits was the Heartbleed Bug [19]. This security vulnerability allowed intruders to read the memory of the systems protected by the vulnerable versions of the OpenSSL. This bug provided the attackers with the opportunity to eavesdrop on communications, steal data directly from services and users and to impersonate them. The problem was that due to a missing bound check on the heartbeat request messages the attacker could request more data than authorized. However, if a deeper analysis to the fault itself is done, it is possible to observe that this vulnerability could be fixed by ignoring the heartbeat request messages that ask for more data than their payload need. In other words, with the addition of some if-statements to prevent the buffer over-read, this vulnerability was removed.

Another famous exploits was the Virtualized Environment Neglected Operations Manipulation, also known as VENON [20]. This vulnerability was present in the virtual floppy drive code used by many computer virtualization platforms. It allowed an attacker to escape from the boundaries of an affected virtual machine. This Virtual machine (VM) escape could give access to the host system and all other VMs running potentially impacting thousands of organizations and millions of end users. However, taking a closer look at the changes made to correct this vulnerability, it is possible to see that it was only necessary a single operation on a variable's value. The vulnerability exploited the memory access that could get out of bounds leading to a memory corruption. So making sure that the index is always bounded, in this case through the use of the modulo operation, the vulnerability was removed.

The Dirty Cow [21] was a security vulnerability on Linux kernel that affected all Linux-based operating systems including Android. The bug exploited a race condition in the implementation of the copy-on-write mechanism in the kernel memory-management. With the right timing, an unprivileged local user could use this flaw to gain write access to otherwise read-only memory mappings. Although it is a local privilege escalation, it can be used in conjunction with other exploits allowing remote root access on a computer. Moreover, another important fact regarding this vulnerability is that the attack does not leave traces in the system log. Looking at the bug-fix patch, there is the possibility to realize that it only requires small code changes to correct this security vulnerability. The vulnerability is reduced to a wrong logical expression used in a branch condition and a wrong logical expression used in an assignment. These small changes can be detected and corrected preventing this vulnerability.

It is observed that the commonly used testing techniques may be failing in some cases. These cases that escape and can lead to serious security breach are the cases that the present thesis proposes to deal with.

1.2 Background

The present dissertation aims to complement the existing software testing strategies in order to detect more vulnerabilities. To this end a study of the situations in which a bug creates a security vulnerability was made.

This dissertation starts by applying and evaluating the existing software testing strategies in the state of the art. With each of these strategies, a test suite for each selected program is produced. A test suite can be defined as a collection of test cases. On the other hand, a test case is defined as an input and an expected result used to exercise the program. Our objective is to measure the effectiveness of these test suites in identifying security vulnerabilities. Based on these results, the next step is to complement the strategies in order to obtain better test suites, capable of detecting more security vulnerabilities. Ntafos [22] conducted a study which compared the effectiveness of three testing techniques (random, control-flow and data-flow testing). The experiment used seven mathematical programs with software faults and applied the three testing techniques to detect these faults. This study reveled that the techniques used have limitations which prevent them from discovering all faults. Therefore, it is important to use many different testing techniques and develop new ones. This idea is presented in Figure 1.1. Our goal is to reduce the gap between the total number of faults present in a program and the number of faults detected using the various testing techniques [1].

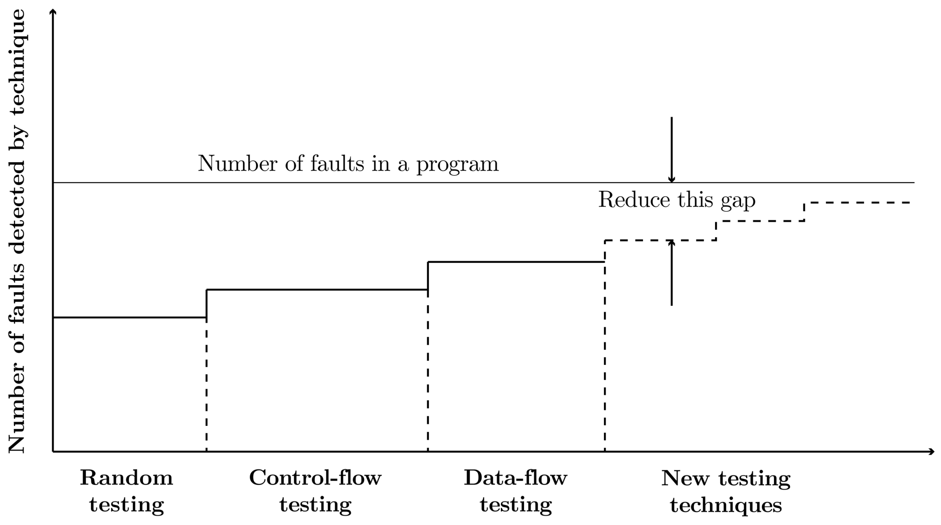

Figure 1.1: Limitations of different fault detection techniques, adapted from [1]

This methodology, inspired by mutation testing, provides the opportunity to understand which types of software vulnerabilities tend to remain undetected to the current strategies of control-flow testing, data-flow testing, etc.

1.3 Scope and Objectives

The core objective of this dissertation is to develop a way to complement the current testing techniques based on knowledge about which types of faults more frequently lead to a security vulnerability. Taking into consideration the main objective, the following goals were developed:

- Study of the conditions in which a software fault leads to a security vulnerability.
- Perform fault injection to emulate representative security vulnerabilities in order to evaluate the commonly used testing techniques and understand where they may fail.
- Develop an approach to improve upon the commonly used testing by finding more vulnerabilities.
- Evaluate the results obtained from the developed approach.

The first goal will be addressed by analyzing real bugs shared in open-source project repositories and manually classifying them. Part of this work has already been done and it is important to extend it to fully understand why some types of faults are more prone to create security vulnerabilities. As said before, not all faults are exploitable, and to accomplish the proposed goals, it is necessary to understand the conditions that lead a bug to create a security vulnerability.

The second goal is the simulation of representative security vulnerabilities in the programs selected. The faults are considered representative because they derive from the study previously conducted on real security faults. The fault injection will help to evaluate the effectiveness of the software testing strategies presented in the state of the art on detecting security vulnerabilities. Each strategy results in a test suite for each of the selected programs. So afterwards, it is possible to assess the type of faults that require more attention.

The third goal aims at the development of a methodology that complements the testing techniques used but focusing more on security vulnerabilities. With the results obtained, it will be possible to understand the types of faults that escape the testing techniques and based on this knowledge, develop a methodology to address these particular faults.

The last goal consists on the validation of the approach presented. A similar evaluation using the same faulty functions will be performed and the results will be analyzed to understand if it complements or not the testing techniques previously used. It is understood that it is almost impossible to create a testing technique capable of detecting all type of faults, the objective consists on complementing the other techniques with a more focused approach.

These objectives together are summarized in the title of this dissertation, software verification targeted for the identification of security vulnerabilities.

1.4 Document Structure

This dissertation is divided in 5 chapters: Introduction, Field study on security vulnerabilities, Testing for security vulnerabilities, Results, Discussion and Conclusion.

The first chapter identifies the theme of this dissertation, the objectives and the motivation behind it. The second chapter, State of the Art, presents the study that was made to deepen the knowledge of the topics of this dissertation. Initially, it is presented a model whose objective is to perform fault classification. Various studies are also shown about software fault injection and the software testing techniques studied for this project. Lastly, some approaches similar to what is intended to be done in this dissertation are introduced. The third chapter presents a review of an earlier study conducted on security vulnerabilities and our deepening of the results. The fourth chapter presents the work done to develop our approach on detecting security related faults. It is also explained in detail the decisions, assumptions and all the steps made to obtain the resulting tool. The results of the application of three testing technique are presented in the next chapter. The sixth chapter, discussion, presents the work plan of this dissertation and the advantages and disadvantages of using the developed tool. The last chapter concludes the dissertation and states the final ideas of the project and the future work.

This page is intentionally left blank.

Chapter 2

State of the art

This chapter presents a technology which extracts valuable information from the defect stream of any software engineering process. Afterwards, software testing will be introduced as well as the commonly used mechanisms and their objectives. It is important to understand the techniques and their limitation to assess their effectiveness. In order to evaluate the detection ratio of these techniques, it was necessary to emulate software faults. Therefore, different fault injection tools will be described in this chapter. Finally, a brief review is presented of studies in the domain so as to understand the current knowledge in the area of this dissertation.

2.1 Software faults and security vulnerabilities

Software faults are programming mistakes that cause a program to malfunction or to produce incorrect/unexpected results. Some faults cause the system to crash, some cause a connection failure, some prevent a user to log in. However, some software faults create data leakage or elevate user privileges, these are security vulnerabilities. Is important to consider that every device has software which contains software faults, thus it is inevitable that it also contains security vulnerabilities [23]. However, as Linus Torvalds said *security problems are just bugs* [24]. A security vulnerability is a software fault that can open a security breach. It is a narrower concept as not all faults are exploitable to the point of creating a security vulnerability, it depends on the type of software, and on the type of fault. They represent a serious concert to programmers and organizations as they represent thousands of millions of euros loss occurring because of what is mostly a preventable problem. Also, with the increasing in size and complexity of today's software projects leads to a growing number of security vulnerabilities.

2.2 Fault Classification

There has been a number of studies on the nature of software faults specifically aimed at systematically classifying them by examining software patches and defect corrections. An important contribution to promote this study of software faults is the Orthogonal Defect Classification (ODC) [25].

The ODC was originally developed in the early 1990s by Ram Chillarege at IBM. This classification is a concept that enables in-process feedback for the developers. It extracts semantic information from the defects that occur through the software development process to create cause-effect relationships. ODC essentially means that a defect is categorized into classes that collectively point to the part of the process that requires more attention.

In the first pilot [26], the classification set was composed of 5 defect types: function, initialization, checking, assignment and documentation. This set provided a sufficient span to explain why the development process had troubles and what could be done to fix it, however, in subsequent discussions and pilots, the model was refined to the current eight classes [25]:

- **Function** - This defect affects significant capability, end-user features, product application programming interface (API), interface with hardware architecture or global structures. It would require a formal design change.
- **Assignment** - An assignment defect indicates a few lines of code, such as the initialization of control blocks or data structure.
- **Interface** - This defect corresponds to errors in interacting with other components, modules, device drivers via macros, call statements, control blocks or parameter lists.
- **Checking** - Addresses program logic that has failed to properly validate data and values before they are used, loop conditions, etc.
- **Timing/serialization** - This defects are those that are corrected by improved management of shared and real-time resources.
- **Build/package/merge** - These errors occur due to mistakes in library systems, management of changes or version control.
- **Documentation** - Errors in documentation that can affect both publications and maintenance notes.
- **Algorithm** - Errors that affect the efficiency or correctness of the task and can be fixed by (re)implementing an algorithm or data structure without the need of a design change.

2.3 Software Fault Injection

The technique of fault injection dates back to the 1970s when it was used at a hardware level [27]. This approach tries to emulate hardware failures into the target system. The system circuit is exposed to some type of interference to produce the fault, and the effect is examined [28].

It is believed that the first technique to inject faults was a pin-level fault injection, which either uses probes to drive current on the target pin to produce voltage level faults or replace the target device by a custom-made socket where digital logical intercepts each target pin. It was primarily used as a test of the dependability of the hardware system and later specialized hardware was developed to extend this technique, such as devices that bombard specific areas of the circuit board with heavy radiation [27].

It was soon found that faults could be emulated using software techniques and this would bring benefits that could be useful for assessing software systems and it was more attractive because they do not require expensive hardware. Collectively these techniques are known as Software Implemented Fault Injection (SWIFI). In the simplest case, SWIFI can confront an interface with randomly generated values. The most complex fault injection tools operate based on detailed failure cause models and can rely on formal specifications to work in a more efficient way to guarantee certain fault coverage criteria.

The injection process can be used as a complimentary technique to the usual software testing mechanisms. While software tests are designed to assert the correct behavior of the system under a representative workload, fault injection asserts the correct behavior under an additional fault load [28, 29, 30].

It is possible to categorize the software fault injection by when the injection takes place:

- Compile-time injection - The program instructions must be modified before the execution. Instead of injecting faults into the hardware of the target system, this technique emulates errors in the source code or assembly code of the target program. The injection generates an erroneous program image and when the system executes it activates the fault.
- Runtime injections - The faults are injected during the program execution using a mechanism to trigger the fault. The commonly used triggering mechanisms are:
  - Time-out - This is the simplest of the techniques. The injection is triggered when the timer reaches a specified time and an interrupt is generated to invoke the fault injection.
  - Exception/trap - Hardware exception or software trap mechanisms are used to generate an interrupt at a specific place in the system code or on a particular event within the system to transfer control to the fault injector.
  - Code insertion - The instructions are added to the target software that allows the fault injection to occur before particular instructions. Unlike the injection before running, this technique performs during run-time and inserts instructions instead of changing the original instructions.
- At the loading time of external components - The injection triggers may be the dynamic binding of external libraries or the adding of other dependencies during run-time.

2.3.1 Software Fault Injection Tools

Although these faults can be manually injected, there is a high probability of introducing unintended defects, so some tools have been developed to parse a program and insert the faults automatically.

Fault Injection Based Automated Testing Environment (FIAT) [31] is an automated real-time distributed accelerated fault injection environment. It was created with the objective of providing suitable tools for the validation process. The FIAT has been designed for the injection of faults that are representative of errors generated by software, hardware or network connection faults in the run time environment of each program. It provides the ability to change where, when, for how long errors will occur. The main objective of this tool is to uncover deficiencies in a system's error detection and recovery mechanism and to guide trade-offs between alternative design enhancements, by providing quantitative evaluations of their relative effectiveness. In its first version, FIAT could emulate faults in user application code and data and also inject faults into messages, tasks and timers.

MODel-Implemented Fault Injection (MODIFI) [32] is a fault injection tool for robustness evaluation targeting behavior models in Simulink. Fault models used by MODIFI are defined using XML according to a specific schema file and the injection algorithm uses the concept of Minimal cut sets (MCS) generation. Firstly, a user defined set of single faults are injected to see if the target system is tolerant against single faults. The fault lading to a failure is stored in a MCS list and removed from the fault space used for subsequent experiments. When all single faults have been injected, the effects of introducing two or more faults at the same time are investigated. The resulting list of MCS is then used to generate test cases to perform the fault injection on the target system.

Fault and ERRor Automatic Real-time Injection (Ferrari) [33] is a tool with the ability to inject transient errors as well as permanent faults so that it can be used to test the effectiveness of concurrent error detection and correction mechanisms. It is implemented to emulate a large number of hardware faults as well as control flow errors and it allows control over the time, location, type and duration of the fault or error. Ferrari consists of four components: An initializer and activator, the user information, the fault injector and the data collector and analyzer. The fault and error injector use software trap and trap handling to change the content of selected registers or memory locations to emulate actual data corruptions. The faults injected can be those permanent or transient that resulting in an address line error, a data line error or a condition bit error.

Fault Tolerance And Performance Evaluator (FTAPE) [34] is a tool that combines a fault injector with a workload generator to encourage a high level of fault propagation. FTAPE has the ability to inject faults into CPU registers, memory and disk systems. Because tolerance mechanisms are present in many different parts of a tolerant system, the faults must be injected into those different parts to measure how the target system behaves. The injector evaluates the instantaneous workload activity to determine the injection time and location that will maximize the fault propagation. The synthetic workload generator is used to provide an easily controllable workload, and it can be specified to exercise the CPU, memory or disk.

Integrated software fault injection environment (DOCTOR) [35] is capable of generating workloads under which system dependability is evaluated, injecting various types of faults with different options and collecting performance and dependability data. A comprehensive graphical user interface is also provided to enable the user to construct complicated fault-injection scenarios. The software-implemented fault injection tool can inject communication faults as well as traditional hardware faults like memory and CPU faults.

Orchestra [36] is a script driven fault injector. The main focus of this tool is the use of experimental techniques to identify specific problems in a protocol or its implementation rather than the evaluation of the system's dependability through statistical metrics. This tool creates a Probe/fault injection (PFI) between two consecutive layers in a protocol stack. Then the PFI layer can manipulate the messages transmitted between the two layers.

Library-level Fault Injector (LFI) [37] is a tool that automates the preparation of fault scenarios and their injection. It is used to simulate exceptional situations that programs need to handle at runtime. This tool automatically identifies the errors exposed by shared libraries doing a static analysis of their binaries. It finds potentially faulty recovery code in the target program binaries and emulates faults between shared libraries and applications layers.

Generic Software Fault Injection Technique (G-SWFIT) [38] consists on finding key programming structures at the machine code-level where high-level software faults can be emulated. It modifies the binary code of software modules by introducing specific changes that correspond to the code that would be generated by the compiler if the fault was in the source code. Emulating software faults at machine-code level brings the advantage of not requiring the source code of the target application to inject the faults. G-SWFIT is a very accurate tool and can be easily ported to practically all types of systems.

Xception [39] focuses on testing and certifying critical systems and validating their reliability. This tool supports and automates the simulation of system failures. It allows testing systems in exceptional situations and scenarios of failures using a predefined set of errors (fault model). The faults are injected with the minimum interference within the target system workload requiring no modification of system source and no insertion of software traps. The tool uses the built-in debugging features of contemporary processors to provide minimum intrusiveness.

ucXception [40] is a tool designed to simplify the software fault injection process. The tool modifies the source code of the target software by altering the abstract syntax tree and producing software patches. The proposed tool formally specifies the most common software fault injection operators and the constraints that should be verified to inject them in a realistic way. It provides a publicly available test suite consisting of source code files and the resulting patches from applying the most representative and frequent software fault operators. It has an improved performance of the fault injection process by compiling only the file in which an emulation operator is applied and linking/installing that file.

2.3.2 Comparison of software fault injection tools

Although there is a reasonably large set of tools, most of these tools inject the faults at system level, creating bit-flips, using software traps or creating CPU faults. However, we want to be able to introduce fault at the source code level to emulate real programmers mistakes. So to accomplish the proposed realistic fault injection, we selected the ucXception tool as it represents a more adequate tool. It alters the source code of the target programs producing patch files that contain each modification representing the bugs that the programmers usually make. That way it is possible to also have access to the faulty code, which helps in the testing phase as white-box techniques were applied to base the test cases in the faulty code and not the original code.

2.4 Software Testing Techniques

Software testing is the process of executing a program or system with the intention of finding errors [41] or involves any activity with the objective of evaluating the software's capability. Software is not like other physical processes in which inputs are received and some outputs are produced. The software differs in the manner in which it fails. Most physical systems fail in a fixed set of ways, however software can fail in many bizarre ways [42].

There are many approaches to software testing, but to effectively test a complex system it is necessary a process of investigation. It is often impossible to test a program in order to find all its errors. This problem has implications for the economic point of view, leaving an open question, as to what would be the strategy that should be adopted for testing.

The three most important techniques that are used for finding errors are black-box testing, white-box testing and grey-box testing.

2.4.1 Black-box Testing

An important testing strategy is black-box, also known as data-driven or input/output-driven testing. This method views the software as a "black-box" — without any knowledge of its internal behavior and structure. Instead, it only examines the fundamental aspects of the system. In this approach the tester does not have access to the source code, using solely the data from the specifications to produce the tests [42, 43].

It is obvious that the more input space covered, the more problems the technique will find and therefore more confidence about the quality of the software. Ideally it would have to exhaustively test all the input space. But as stated above, exhaustively testing is impossible. Another problem is due to the limitations of the language used in the specifications, usually natural language, in which ambiguity is often inevitable. The research in black-box testing main focus is on how to maximize the effectiveness of testing with minimum cost. As it is unfeasible to exhaust the input space, it is common to use partitioning in order to test exhaustively a subset of the input space [41].

Some important types of black-box testing techniques are the following.

- Random Testing - Random testing is a form of functional testing that is useful when the time needed to write and run the tests is too long or the complexity of the problem makes it impossible to test every combination. Another advantage is that even when if it doesn't find many defects per time interval, it can be performed without manual intervention [43, 44, 45].

  Considering the following function.

  ```
  int myAbs(int x) {
      if (x > 0) {
          return x;
      }
      else {
          return -x;
  ```

```
        }
    }
```

In order to test this function it was necessary to generate a random set of input, for example 1,27,-34,0,-93.

- Equivalence Partitioning - This method divides the input data into partitions of data from which test cases can be created. An equivalence class is created by the inputs for which the behavior of the system is expected to be identical. Typically, an input condition is either a numeric value, an array of values, a set of related values or a Boolean condition [43, 44]. For example, testing a password field that accepts a minimum of six characters and maximum of ten characters. It is not possible to test all the possible number of characters because the possibilities are infinite. To address this problem, the possible number of characters was divided into sets called equivalence partitions or equivalence classes. In this case the input was divided in three classes. From zero to six characters, the system should not accept, between six and ten characters, it should accept, and between ten and more characters, the system should not accept. Then, it is necessary to pick at least a value from each partition for testing [43, 44, 46].

- Boundary Value Analysis - This technique explores the system tendency to fail on boundary values. Due to programmers often making mistakes on the boundary of the input domain, the boundary values must include the maximum, minimum, inside and outside boundaries, typical values and error values [43, 44]. The basic idea in boundary value testing is to select input variable values at their:

  - Minimum
  - Just above the minimum
  - A nominal value
  - Just below the maximum
  - Maximum

- Fuzzing - Fuzz testing is a technique which is used for finding implementation bugs by inputting malformed/semi-malformed data in an automated or semi-automated fashion. It is used to find bugs associated with assertion failures and memory leaks. It is also used to test for security problems in software. The main limitation with this method is that it generally only finds very simple faults [43, 44]. For example, this can be performed by shifting random blocks of bits through the file of existing inputs.

- Cause-Effect Graph - This testing technique begins by creating a graph and establishing the relation between the effect and its causes. The cause is the input condition, and the effect is a sequence of computations to be performed. The graph shows the nodes representing the causes on the left and the nodes representing the effects on the right side. There may be intermediate nodes in between that combine the inputs using logical operators such as AND and OR [43, 44]. Taking as an example a software application that reads two characters and, depending of their values a message is printed or a file is updated.

  - The first character must be an *'A'* or a *'B'*.
  - The second character must be a digit.
  - If the first character is an *'A'* or *'B'* and the second character is a digit, the file is updated.

- If the first character is incorrect (not an *'A'* or *'B'*), the message X is printed.
- If the second character is incorrect (not a digit), the message Y is printed.

Causes:

- C1 - The first character is *'A'*
- C2 - The first character is *'B'*
- C3 - The second character is a digit

Effects:

- E1 - The file is updated — (C1 OR C2) AND C3
- E2 - The message X is printed — (NOT C1 AND NOT C2)
- E3 - The message Y is printed — (NOT C3)

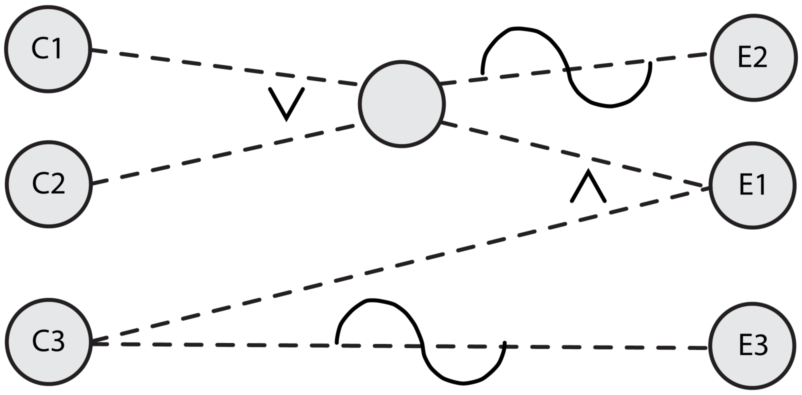

Figure 2.1: Example of a cause-effect graph, adapted from [2]

From the graph, it is possible to create the table with the test cases.

Table 2.1: Example of decision table, adapted from [8]

Nodes	**TC 1**	**TC 2**	**TC 3**	**TC 4**	**TC 5**	**TC 6**
C1	1	0	0	0	1	0
C2	0	1	0	0	0	1
C3	1	1	0	1	0	0
E1	1	1	0	0	0	0
E2	0	0	1	1	0	0
E3	0	0	0	0	1	1

For the test case 1 (TC 1) it is necessary to validate that the system updates the file when the first character is *'A'* (C1) and the second character is a digit (C3) and the outcome should be E1, the file updated. For the test case 2 (TC 2) it is necessary to validate that the system updates the file when the first character is *'B'* (C2) and the second character is a digit (C3) and the outcome should be E1, the file updated. In a similar fashion, it is possible to create the remaining test cases.

- All Pair Testing - It is a test design technique in which test cases are designed to execute all the possible discrete combinations of each pair of input parameters. Using carefully chosen test vectors, this can be done faster than executing all combinations of all parameters [43, 44]. If there are 'n' parameters, each with 'm' different values then between each two parameter we have n * m pairs. Assuming we have an application to test which has a simple list box with 10 elements, a checkbox, a radio button and a text box. The text box can only accept values between 1 and 100. So the values that each one of the GUI objects can take are the following.

– List Box - 0,1,2,3,4,5,6,7,8,9
– Check Box - Checked or Unchecked
– Radio Button - On or Off
– Text Box - Values between 1 and 100

The number test cases using Cartesian method: 10*2*2*100 = 4000. So the idea of all pair testing is to reduce the number of test cases. It is possible to consider that the list box values as 0 and others as 0 is neither positive nor negative. The values in the text box can also be reduced into three types of inputs, valid integer, invalid integer or alpha character. Now, the number of test cases needed are calculated using a non-combinatorial testing technique: 2*2*2*3 = 24. However, using the all pair testing technique it is still possible to reduce the combinations. Table 2.2 presents the test cases created.

Table 2.2: Example of test cases created using all pair testing, adapted from [9]

Text Box	**List Box**	**Check Box**	**Radio Button**
Valid Int	0	Check	On
Valid Int	Others	Uncheck	Off
Invalid Int	0	Check	On
Invalid Int	Others	Uncheck	Off
Alpha Character	0	Check	On
Alpha Character	Others	Uncheck	Off

• Orthogonal Array Testing - Orthogonal Array Testing (OAT) is a testing technique applied when the domain of inputs to the system is relatively small but too large to accommodate exhaustive testing. It is a technique similar to all pair testing but instead of using each combination of parameters, an orthogonal array is used to choose just a subset of these combinations. This enables to design test cases that provide maximum test coverage with reasonable number of test cases [43, 44]. For example a web page that has three distinct sections (Top, middle and bottom) that can be individually shown or hidden from a user. There are three factors (Top, middle and bottom), each of them with two levels (Shown or hidden). If a conventional testing technique is chosen, 2 * 3 = 6 test cases are needed.

Table 2.3: Example of conventional testing technique, adapted from [10]

Test Case	**Scenarios**	**Visibility**
1	Hidden	Top
2	Shown	Top
3	Hidden	Middle
4	Shown	Middle
5	Hidden	Bottom
6	Shown	Bottom

If Orthogonal array testing is chosen, it requires four test cases as shown below.

Table 2.4: Example of Orthogonal array testing technique, adapted from [10]

Test Case	**Scenarios**	**Visibility**
1	Hidden	Top
2	Shown	Top
3	Hidden	Middle
4	Shown	Middle

- State Transition Testing - It is another testing technique which is useful when testing state machines and navigation of graphical user interface. It is defined as a method in which input condition changes cause a state change in the application under test. The main advantage of this technique is to generate a state transition diagram, so the tester can understand the system behavior more efficiently [43, 44]. Taking the ATM system as example where the user enters the password to access his/her account, and if he/she makes a mistake three times the account will be locked.

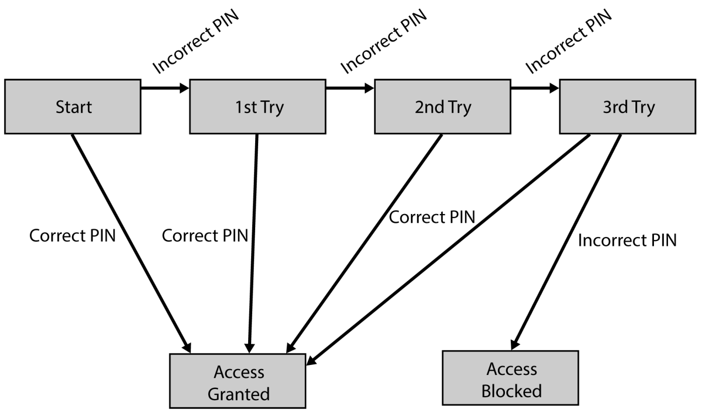

Figure 2.2: Example of a state transition diagram, adapted from [3]

As presented in the diagram, whenever the user enters the correct PIN the state moves to Access granted, and if he/she enters an incorrect password the state changes to a next try and if he/she repeats it two more times it reaches a blocked state.

Table 2.5: Example of a state transition table, adapted from [3]

States	**Correct PIN**	**Incorrect PIN**
S1 - Start	S5	S2
S2 - 1st attempt	S5	S3
S3 - 2nd attempt	S5	S4
S4 - 3rd attempt	S5	S6
S5 - Access Granted	-	-
S6 - Account blocked	-	-

When the user enters the correct PIN the state is changed to S5 which is access granted. If the user enters a wrong password he/she is moved to next state and if the same thing happens three times, he/she will reach the account blocked state. The tests can be designed to test every transition or to cover all pairs of two valid transitions shown in the diagram.

2.4.2 White-box Testing

Another testing strategy, white-box also known as logic-driven testing, allows the examination of the internal structure of the target software. It is one of the most important software testing techniques and is typically very effective in finding programming and implementation errors. White-box testing is based on the analysis of internal workings and structure of a piece of software to derive test cases [43, 47].

This technique causes every statement in the program to execute at least once with the idea that if all the possible paths of control flow are executed then possibly the program has been completely tested. To help with the white-box testing is often created a control flow diagram of the target software. In this diagram each node represents a segment of statements that execute sequentially, possible terminating with a branch condition. Each represents a transfer of control between segments [41].

Some important types of white-box testing techniques are the following.

- Control Flow Testing - Control flow testing can be applied to almost all software and is effective for most software. It is a testing strategy that uses the program's control flow as a model to design the test cases. It favors more but simpler paths over fewer but complicated path. Studies show that control-flow testing catches half of all bugs caught during unit testing, this happens because most of the bugs can result in control flow errors and therefore a wrong functionality of the software could be caught by control flow testing [43, 47].

  The adequacy and completeness of the set of test cases is measured with a metric called coverage. Some coverage methods are the following.

  – Statement Coverage - Statement coverage is a measure of the percentage of the statements that have been executed at least once by test cases. Less than 100% of statement coverage means that not all lines in the source code have been exercised. The main drawback of this technique is that it does not test the false conditions in the source code.

  – Branch Coverage - A stronger coverage criterion is the branch coverage. This is a measure of the percentage of the decision point of the program has been evaluated as both true and false in test cases.

  – Condition Coverage - A even stronger criterion is condition coverage. It is a measure of percentage of Boolean sub-expressions of the program that have been evaluated as both true and false in the test cases.

  Using the following control-flow graph as example.

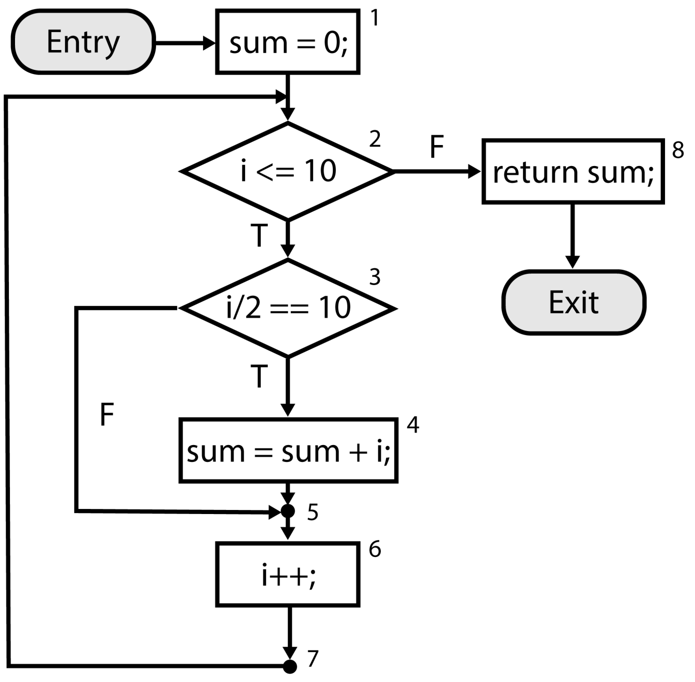

Figure 2.3: Example of a control flow graph, adapted from [4]

For the statement coverage it is necessary that every statement in the program has been executed at least once:

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 8$

For the decision coverage it is necessary that every statement in the program has been executed at least once and every decision in the program has taken all possible outcomes at least once:

$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 8$

For the condition coverage it is necessary that every statement in the program has been executed at least once and every condition in each decision has taken all possible outcomes at least once. Assuming we want to test the following code.

```
if ( (A || B) && C ) {
    /* instructions */
}
else {
    /* instructions */
}
```

A, *B* and *C* represent atomic boolean expressions (i.e. not divisible in other boolean sub-expressions). In order to ensure condition coverage for this example, *A*, *B* and *C* should be evaluated at least one time as *true* and one as *false*. So, in this example, we need three tests to ensure condition coverage.

A = true and B = not evaluated and C = false

A = false and B = true and C = true

A = false and B = false and C = not evaluated

- Basis Path Testing - Basis path is a technique first introduced by Tom McCabe [48]. It analyzes the control flow graph of the target program to find a set linearly independent paths of execution. This method usually uses McCabe' cyclomatic complexity to determine the number of linearly independent paths and then generates test cases for each path. Basic path testing guarantees complete branch coverage, all the edges of the control flow graph, but it will not cover all the possible paths of the graph [43, 47]. Taking the following graph as example [5].

  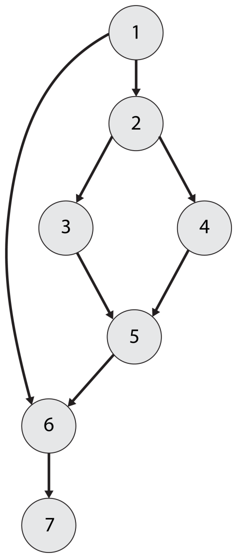

  Figure 2.4: Example of a control flow graph, adapted from [5]

  In this example there are 3 paths or condition that need to be tested.

  - Path 1: 1,2,3,5,6,7
  - Path 2: 1,2,4,5,6,7
  - Path 3: 1,6,7

- Data Flow Testing - Data flow testing uses a model of the flow of data connected with the flow of control to understand how the program variables are defined and used. It is used to check every data object that has been initialized prior to its use and all the defined objects that have been used at least once. The method divides the variable occurrences in two, definitions and uses. The definitions are occurrences where a variable is given a new value. The uses are occurrences where a variable is not given a new value, but where the variable is the predicate portion of a decision statement (P-uses) or where the variable appears on the right side of an assignment statement or an output statement (C-uses) [43, 47, 49].

  Using the following piece of code as example [49, 50].

```
1. read x;
2. if( x > 0)                        (1,(2,t),x), (1,(2,f),x)
3.    a = x + 1                      (1,3,x)
4. if (x <= 0){                      (1,(4,t),x), (1,(4,f),x)
5.    if (x < 1)                     (1,(5,t),x), (1,(5,f),x),
                                     (6,(5,t),x), (6,(5,f),x)
6.       x = x + 1; (go to 5)        (1,6,x), (6,6,x)
7.    else
8.       a = x + 1                   (1,8,x)
```

```
        }
    9. print a;                      (3,9,a), (7,9,a)
```

It is possible to create test cases according to a certain test criteria. One example is the all definitions coverage. In this testing coverage criteria it is necessary to create test cases that include a path from every definition to some corresponding use (c-use or p-use), in this case the definitions are the following.

(1, (2, f), x), (3, 9, a), (6, 6, x), (7, 9, a)

since they represent all occurrences of a value being attributed to a variable. If we want to cover all the computation uses (c-uses) we need test cases that include a path from every definition to all of its corresponding c-uses, in this case the c-uses are the following.

(1, 3, x), (1, 6, x), (1, 8, x), (3, 9, a), (6, 6, x), (6, 8, x), (7, 9, a)

They represent all the statements where a variable is used to compute a value for output or for defining another variable. To cover all the predicate uses (p-uses) we need to create tests cases that cover a path from every definition to all of its corresponding p-uses, in this case the p-uses are the following.

(1, (2, t), x), (1, (2, f), x), (1, (4, t), x), (1, (4, f), x), (1, (5, t), x), (1, (5, f), x), (6, (5, t), x), (6, (5, f), x)

There are other types of coverage such as all c-use some p-use coverage. It is similar to the all c-uses coverage with the difference that if a definition has no c-use then the test cases include a path to some p-use. The same happens with the all p-use some c-use coverage, if a definition has no p-use then the test cases include a path to some c-use. There is also the all uses coverage criteria where the test cases must include a path from every definition to each of its uses, including both c-uses and p-uses.

- Loop Testing - Loop testing is another type of white-box testing which exclusively focuses on the validity of loop construct. It can reveal loops initialization problems and also reveal performance bottlenecks [43, 47]. For example, a simple loop is tested in the following ways [51].
  - Skip the entire loop
  - Make 1 pass through the loop
  - Make 2 pass through the loop
  - Make *m* pass through the loop where *m* ¡ *n*, *n* is the maximum number of passes through the loop
  - Make *n*, *n*-1; *n*+1 pass through the loop where *n* is the maximum number of allowable passes through the loop.
- Mutation Testing - Mutation testing is a powerful, yet computationally expensive, technique for unit testing software. It provides a testing criterion called the mutation adequacy score which can measure the effectiveness of a test set in terms of its ability to detect faults [52].

The general principle behind mutation testing is to emulate faults into the target software that represent the mistakes that programmers usually make. These faults are deliberately emulated into the program to create a set of faulty programs called mutants, each containing a different change. To determine the quality of a test set, these mutants are executed, if the resulting output is different from the output of running the original program for any test case, then the fault in the mutant was detected. One of the outcomes of this process is the mutation score, which is an indicator of the quality of the input test set. The mutation score is defined by the quotient between the number of detected faults and the total number of injected faults [52]. Taking this piece of code as example :

```
int myAbs(int x) {
    if (x > 0) {
        return x;
    }
    else {
        return -x;
    }
}
```

We can create mutants making small changes in the original code such as replacing arithmetic operators or removing statements, for example:

```
int myAbs(int x) {
    if (x < 0) {  // Replacing the > with <
        return x;
    }
    else {
        return -x;
    }
}

int myAbs(int x) {
    if (x %% 0) {  // Incorrect syntax
        return x;
    }
    else {
        return -x;
    }
}
```

Once all the mutants are created they are subjected to the test data-set. If the data-set detects all mutants, it is effective. Otherwise we include more or better test data. It is not necessary for each test in the test suite to detect all mutants but together they should detect all [8].

2.4.3 Grey-box Testing

Grey-box testing technique increases the testing coverage by allowing to focus on all the layers of any complex system through the combination of white-box and black-box testing. It still tests the software as a black-box, but supplements the work by taking a peek (not a full look, as in white-box testing) at what makes the software work. This involves having access to the internal structures and algorithms to design the test cases, but testing at the black-box level [43, 53].

Different forms of grey-box testing techniques are the following.

- Orthogonal Array Testing - A statistical testing technique first developed by Genichi Taguchi [54]. The black-box technique will not provide sufficient testing coverage. A black-box team cannot understand the underlining infrastructure connections between servers and legacy systems, however a grey-box testing team will have the necessary knowledge to elaborate a testing net that can be set-up and implemented [11]. This technique is used to reduce the number of combinations of inputs but sill maximize the coverage. The technique uses an array of values, where each column represents a variable, which can take a set of values called levels. Each row represents a test case and then the values are combined pair-wise to create an efficient, concise test set. Supposing we are testing a software web application. We need to test different operative systems and different browsers [11, 43, 55]. In table 2.6 all the conditions we want to test are shown.

  Table 2.6: Example with three factors and three levels, adapted from [11]

	OS	**Browser**	**Function**
	Factor 1	**Factor 2**	**Factor 3**
Level 1	Linux	Chrome	Library
Level 2	Mac OS	Internet Explorer	Scheduling
Level 3	Windows	Firefox	Personal Information

  With this example the tester would need to create $3^3 = 27$ test combinations. Using the orthogonal array testing we reduce the total number of test cases drops from 27 to nine.

  Table 2.7: Table of test cases, adapted from [11]

Test Case	**Factor 1**	**Factor 2**	**Factor 3**
1	1	1	3
2	1	2	2
3	1	3	1
4	2	1	2
5	2	2	1
6	2	3	3
7	3	1	1
8	3	2	2
9	3	3	2

- Regression Testing - Regression testing is a technique executed after making a functional improvement or a repair to the software. Its objective is to determine if the change made has regressed the other features of the program [43].

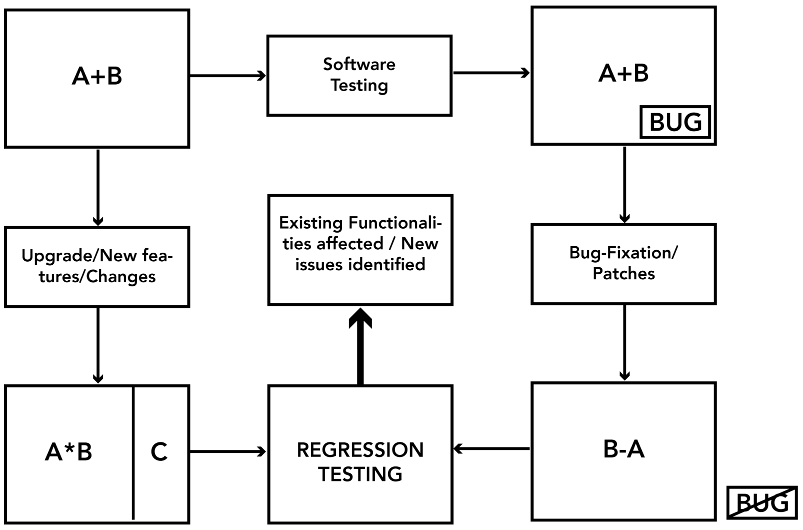
A+B
Software Testing
A+B
BUG
Upgrade/New features/Changes
Existing Functionalities affected / New issues identified
Bug-Fixation/ Patches
A*B
C
REGRESSION TESTING
B-A
~~BUG~~

Figure 2.5: Regression testing diagram, adapted from [6]

We have a software application with functionalities A and B. Testing the application reveals a bug. Patches and bug-removal processes are applied to the code in order remove or resolve the identified bug. However, the process may affect the existing functionalities: B-A. So it is necessary to execute regressing testing to evaluate if the existing functionalities have been impaired or not with the bug removal.

Another example is when we want to upgrade or add a new feature C and the existing functionalities of the software application have been changed with the new feature: A*B. In this case, regressing tests are executed to find if the existing functionalities have been affected or not.

- Pattern Testing - This technique is best executed when data from system's previous defects are analyzed. The analysis will include the reasons behind the defect to determine why the failure happened. This information is valuable as future design of test cases will use this information to prevent similar faults [11, 43].

- Matrix Testing - This technique starts by defining all the variables that exist in the target program. Then each variable will have an inherent technical risk and business risk and can be used with different frequencies during its life cycle. The objective is to eliminate uninitialized and unused variables by identifying the ones used by the program. For example, for this type of testing it is important to summarize the information in two types of tables as the following [11, 43].

Table 2.8: Client name and address input-recall testing, adapted from [12]

Element	**Data type**	**Min length**	**Max length**	**Create**	**Read**	**Update**	**Delete**
Name	String	2	20	Yes	Yes	Yes	No
Surname	String	2	25	Yes	Yes	Yes	No
Address	String	2	45	Yes	Yes	Yes	Yes
City	String	2	45	Yes	Yes	Yes	Yes

Table 2.9: Summarized information of Table 2.8, adapted from [12]

Function	**Technical Risk**	**Frequency**	**Business Risk**
Update Record	Low	Medium	Low
Save Record	Medium	High	High
Delete Record	High	Medium	Medium

With the information from table 2.8 and table 2.9 the tester can decide which technical and business aspect of the code, in this case saving and deleting records, require a more careful testing.

2.5 Related projects

Du [56] described an approach for vulnerability testing using fault injection. Usually, security testing uses penetration testing and formal methods, but the effectiveness of these methods can be influenced by the team that performs the analysis. The authors noticed that most security faults are triggered because of a faulty interaction with the environment. In order to prove this, environment faults were injected into the target system and observed their behavior. Nowadays, the importance of a safe product has been growing more and more, so this work is focused on the evaluation of the software capacity to detect faults that might lead to security violations. The authors chose the fault injection because it allows the emulation of defects independently of how they could occur in practice. The objective of this paper is to create a way to overcome the limitation of when the testers' knowledge about the environment is reduced.

Loise et al.[57] designed security aware mutation operators to support mutation analysis. As the operators used by the Java mutation testing tools are restricted to simple faults they may not suitable to exercise security related aspects of the applications. Therefore, the authors designed 15 security mutation operators for Java based on common security vulnerability patterns. Using the FindBugs tool they showed that the traditional mutation operators from the PIT mutation testing engine fail to introduce vulnerabilities. So authors extended it so it supports the traditional and the new mutation operators. They also showed that their security aware mutation operators are applicable on open source projects, providing evidence that mutation analysis can be applied in security testing activities.

Jimenez et al.[58] performed an analysis focused on the vulnerabilities of two security critical open-source software systems. The main objective was to increase the understanding of the aspects that characterize the vulnerabilities on each project, such as location, type and criticality. In order to perform this, the authors analyzed 863 vulnerabilities and used the Common Weakness Enumeration to categorised them. They found that it is important to make project specific approaches focusing on specific types of vulnerabilities, since the different vulnerability types have different profiles. The results also show that the majority of the vulnerabilities are located on up to 4 directories of the analyzed projects and only a few types are critical. The results also suggest that these two aspects are related.

Durães and Madeira [13] also present a significant study on fault emulation. This paper addresses the emulation of software faults for software reliability. The authors analyzed an extensive collection of real software faults according to the Orthogonal Defect Classification [25]. In a second step the faults were grouped according to the nature of the defect, that is for each ODC class a software fault is characterized by one programming language construct that is either missing, wrong or superfluous. In the resulting data set we can see that 67.6 percent of all the faults collected can be represented by a small set of defects. With the data set collected they made a more practical approach using a tool called G-SWIFT [38]. The objective was to evaluate the tool's accuracy on finding key programming structures at the machine code-level where high-level software faults can be emulated. In the end, the results were usually simple programmer mistakes that are responsible for most software failures. It was also shown that a small subset of specific faults types is dominant regarding fault occurrence, being these "good" candidates for the emulation of software faults.

A similar approach was carried out by Basso et al. [59]. The authors analyzed a 574 faults in order to understand if the fault representativeness of Java applications follows the same pattern as the software faults in C applications. They selected the Java programming language because it is widely used in software applications, including web-based systems. In order to perform a complete classification, the authors refined the classification presented by Durães and Madeira [13] in order to accommodate new fault types. These fault types were created because as Java has different characteristics and is object-oriented, unlike C, there are some faults that cannot be classified in accordance with the methodology proposed by Durães and Madeira [13]. The results show that the most frequent Java faults correspond to the most frequent C faults, which means that the mistakes made by programmers follow a similar pattern, independently of the programming language.

2.6 Summary

To sum up the findings of the conducted research, there are a few software fault injection tools but most of them emulate the faults at the system level becoming difficult to create a representative emulation of programmers mistakes. However, one of the studied tools (ucXception) creates the faults in the form of patches files, allowing us to control the type of faults to inject and how they are injected.

It was also conducted a study about most known testing techniques. Some of these techniques are applied to certain types of situations, like the regression testing technique only applied to a more complex software with various functionalities. Ideally, we should use a black-box and white-box technique to try to generalize our study. It is also important to understand how they exercise the code to find the defects. With this information we can comprehend where they fail to, in the end, focus our approach based on these limitations.

Loise et al.[57] provided evidence that mutation analysis can be applied to large real-world projects. The authors with their study also reveled that a certain type of vulnerabilities are prevalent in open source projects. Jimenez et al.[58] showed that the bugs on the Linux kernel and in the OpenSSL are very located, restricting themselves to a few directories and to a few types of faults. Durães and Madeira [13] present a field study on software faults that resulted in a fault classification based on ODC classes. They also showed that the most frequent faults in C correspond to a small subset of fault types. Basso et al. [59] presented a similar approach based on the Java programming language. They used the classification system presented by Durães and Madeira, and came to very similar conclusions. Only a small subset of faults characterizes the mistakes that programmers make, and this subset is similar in both languages. This demonstrates that the faults are related to the programmers and not the language used.

This page is intentionally left blank.

Chapter 3

Field study on security vulnerabilities

This chapter presents a short review of a field study conducted on security vulnerabilities and the results obtained. In addition, it is also presented the expansion of the results based on the data set created in the field study to understand the conditions in which a software fault leads to a security vulnerability.

3.1 Previous field study overview

We took advantage of a field study [7] that we previously conducted to clarify the most common programming mistakes that result in security vulnerabilities. For each analyzed software projects (Xen hypervisor, Linux kernel and the OpenSSH tool) a specific software version was selected. Then the patch files were collected from public vulnerabilities databases, starting from the most recent to the oldest vulnerability that affected the selected versions. Each patch was analyzed to confirm that the modification was only altering C code and not source code macros. This resulted in 147 vulnerabilities, corresponding to all the vulnerabilities in the OpenSSH tool and the majority of vulnerabilities for the other two projects. Finally, we manually mapped the modifications carried out to correct the vulnerability onto emulation operators described by Durães et al. [13]. These emulation operators are similar to the operators used in mutation testing with some differences. The emulation operators are defined as minimal code changes in order to emulate a unitary programming mistake. They are based on field observation of real faults and not synthetically generated like the mutation operators.

Firstly, the study analyzed how the vulnerabilities are distributed in terms of number operators required to reverse the fix, number of functions and the number of files involved. The results are presented in Figure 3.1. As it is possible to observe, 64 out of 147 vulnerabilities (43.5%) consist on the need of a single operator to emulate the vulnerability. This means that it is only necessary one emulation operator in order to create this type of vulnerability. In addition, it shows that in 30 of the cases (20.4%) the vulnerability affected a single function but was composed of multiple operators. This means that to correct these vulnerabilities, it was only required to change the code of one function. On the other hand, to emulate these vulnerabilities multiple fault operators were injected. In the remaining cases, 20 vulnerabilities affected multiple functions but all the functions changed were located in a single file. The remaining 33 vulnerabilities involved corrections on multiple files. It is important to highlight that the majority of the vulnerabilities affected a single function, either using one or multiple operators. These cases constitute a total of 63.9% of all the vulnerabilities analyzed. This information is important because the emulation involving a single function is less complex than doing so for multiple functions, and far less complex that those involving multiple files [7].

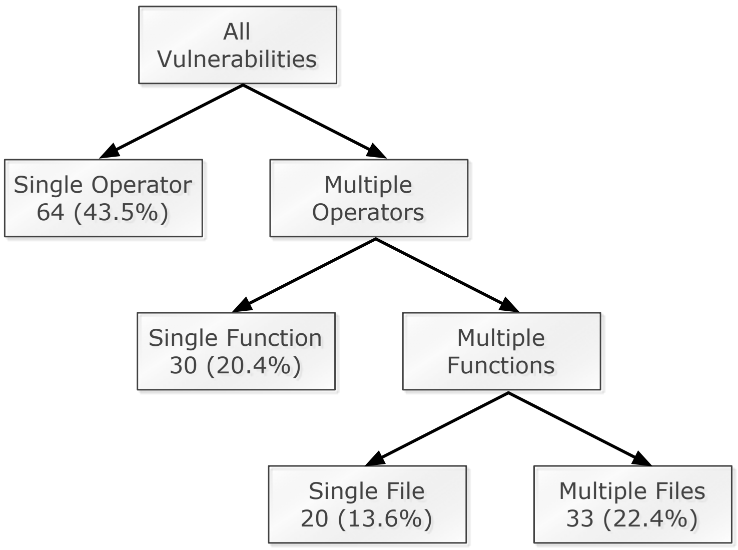
All Vulnerabilities
Single Operator 64 (43.5%)
Multiple Operators
Single Function 30 (20.4%)
Multiple Functions
Single File 20 (13.6%)
Multiple Files 33 (22.4%)

Figure 3.1: Distribution of emulation operators across multiple functions and multiple files, from [7]

We also analyzed how many emulation operators are needed to emulate each vulnerability. Even though we know that 63.9% of the vulnerabilities affect a single function, it is important to understand how many operators they usually consist of. The results are shown in Figure 3.2. It is worth noting that if an operator appears more than once in a single vulnerability all the occurrences are counted in the histogram.

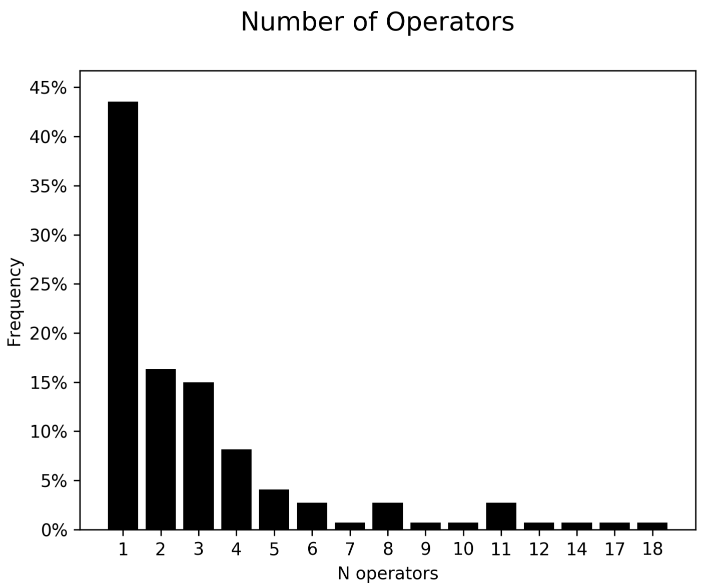

Figure 3.2: Absolute frequency of vulnerabilities over the number of operator instances necessary to emulate them, from [7]

As it is possible to observe in Figure 3.2, the majority of the vulnerabilities consist of up to three programming mistakes. A single operator is sufficient in 43.5% of all defects. The use of two and three operators represent 16.3% and 15.0% of the vulnerabilities, respectively. It is important to highlight that approximately 74.8%, nearly three-fourths of the vulnerabilities, are composed of up to three programming mistakes.

We also collected the types of defect that are relatively frequent in software vulnerabilities. In Table 3.1 it is presented a detailed analysis of the actual programming mistakes found in software projects. It presents the number of vulnerabilities in which each operator was found. It is worth noting that if an operator appears multiple times in the same vulnerability, it is counted as a single vulnerability.

Table 3.1: Most frequent types of programming mistakes that cause software vulnerabilities

Operator	**Description**	**Type**	**#Faults**
MIFS	Missing if construct plus statements	Algorithm	57 (38.8%)
MFC	Missing function call	Algorithm	32 (21.8%)
EFC	Extraneous function call	Algorithm	16 (10.9%)
WLEC	Wrong logical expression used as branch condition	Checking	12 (8.2%)
EIFS	Extraneous IF construct plus statements	Algorithm	10 (6.8%)
MLOC	Missing OR EXPR in expression used as branch condition	Checking	10 (6.8%)
MLAC	Missing AND EXPR in expression used as branch condition	Checking	10 (6.8%)
WALR	Wrong algorithm - code was misplaced	Algorithm	9 (6.1%)
MVAV	Missing variable assignment using a value	Assignment	9 (6.1%)

In Table 3.1 the fault operators that are more commonly related with security vulnerabilities are presented. As it is possible to observe, the Missing if construct and surrounded statements (MIFS) and Missing function call (MFC) operators are the most prone to create a security vulnerability. Missing some function call or if-statement may lead to an incomplete memory cleaning or the lack of a *NULL* verification, therefore these operators can easily lead to a security vulnerability. Also, three of the most common operators are related to a missing or incorrect use of expressions in branch conditions (Missing OR sub-expr in expression used as branch condition (MLOC), Missing AND sub-expr in expression used as branch condition (MLAC) and Wrong logical expression used as branch condition (WLEC)). Having a fault in a branch condition can lead the system through an erroneous execution path putting the system in a vulnerable state which can be exploited by an attacker.

These results are important because the injection of security vulnerabilities involves different operators than the injection of software fault. In order to understand the differences, we analyzed a similar approach made by Durães *et al.* [13] on software faults. The authors analyzed more than 650 real software faults and identified the most frequent operators. The results are shown in Table 3.2.

Table 3.2: Most frequent fault types occurring in software, from [13]

Operator	Description	Type	#Vulnerabilities
MIFS	Missing if construct plus statements	Algorithm	71 (10.6%)
MLAC	Missing AND EXPR in expression used as branch condition	Checking	47 (7%)
MFC	Missing function call	Algorithm	46 (6.9%)
MIA	Missing if construct around statements	Checking	34 (5.1%)
MLOC	Missing OR EXPR in expression used as branch condition	Checking	32 (4.8%)
MLPA	Missing small and localized part of the algorithm	Algorithm	23 (3.4%)
WLEC	Wrong logical expression used as branch condition	Checking	22 (3.3%)
MVAE	Missing variable assignment using an expression	Assignment	21 (3.1%)
MFCT	Missing functionality	Function	21 (3.1%)

As it is possible to observe in Table 3.2, the majority of the software faults are related with missing constructs. As humans we make mistakes, therefore we can forget a functionality or forget to make a function call. Also this type of faults is the one that represents a bigger difficulty to the testing techniques. The common testing techniques exercise the code based on its structure, however, if something is missing, the technique cannot exercise it as it is not there.

In order to make the comparison between the two studies easier, we made a Venn diagram showing the most common operators associated with software faults and the most common operators associated with security vulnerabilities. The diagram is presented in Figure 3.3.

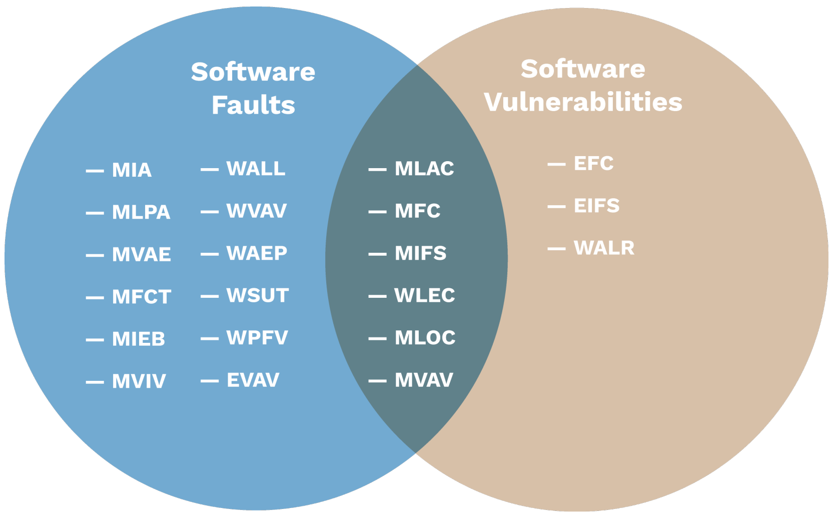

Figure 3.3: Most common operators associated with software faults and security vulnerabilities

It is possible to observe in Figure 3.3 that some operators that cause software faults are also associated with security vulnerabilities. This can be explained by the fact that even though the objective of our analysis [7] was focused on the faults that cause security vulnerabilities, they are still software faults. Therefore, it is expected to find some operators that are common in both sides.

As it is also possible to observe, our study [7] shows new operators which have not been included in the emulation of software faults like, the Extraneous function call (EFC), the Extraneous IF construct plus statements (EIFS) and the Wrong algorithm — code was misplaced (WALR) types of defect. Unlike the rest, the extraneous operators usually do not compromise the correct behavior of the software, otherwise they would be detected by common software testing techniques. Instead they execute code that is unnecessary to the correct outcome but may contain faults. The same happens with the WALR operator. A change in the order of the statements may not cause any change to the output of the software but, may cause the software to enter into a faulty state which can open a security breach.

The MIFS and MFC operators are the most common among the software faults and also the most prone to create a security vulnerability. Missing some kind of expression may lead to an incomplete memory cleaning or to the lack of a *null* check, therefore, these operators can easily lead to a security vulnerability.

From the software vulnerabilities side, it is possible to observe that four of the operators most common types of faults are related to if-statements (MLAC, MIFS, WLEC and MLOC). Missing or performing a wrong check on a variable or a memory address can lead to incorrect executions in which an adversary may exploit the lacking coverage. Also two of most common types of faults are related to function calls (MFC and EFC).

3.2 Study expansion

Based on the results obtained by the field study, it is possible to understand which type of faults are more prone to create a security vulnerability, however, it is still important to understand which type of functions are more likely to contain these faults. In order to accomplish this, we decided to deepen our study [7]. Thus, from this point onward, it is presented the work carried in the scope of this dissertation.

From our previous study, we created a data set with information about the fault injection operators required to revert the correction of each vulnerability, the file and function where the vulnerability was and the number of files and the number of functions it affected. With access to this information and since the three projects are open source, we were able to collect the source code from each function before the vulnerability correction. Based on the code from the faulty functions, we measured the cyclomatic complexity and the number of line of code to understand the type of functions that contain vulnerabilities. We decided to focus the analysis on the vulnerabilities that affected a single function, which covers up to two-thirds of all the vulnerabilities. Out objective is to develop an approach more focused on the most frequent faults, increasing the probability of achieving good results.

Vilela et al. [60] conducted a study about the evaluation of simple metrics as good predictors of software faults. In this study, the authors concluded that the number of lines tend to be a good indicator of the presence of faults. So we collected all the functions analyzed in our study and searched the original source code before the correction. In a first approach, we analyzed the number of non-commented lines of code (nloc) of these functions. The results are presented in the graph of Figure 3.4. The graph presents the probability of the presence of a software fault according to the number of lines.

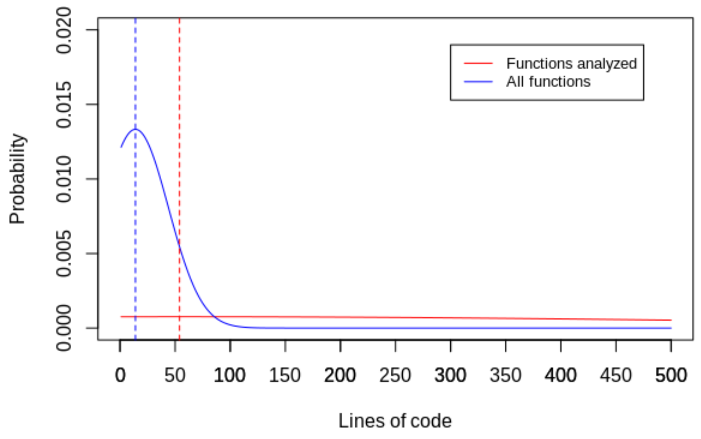

Figure 3.4: Lines of code distribution of the functions analyzed and all functions

The functions analyzed represent all the functions that were present in the data set and contain security vulnerabilities. As it is possible to observe in Figure 3.4, the functions with the security vulnerabilities have, in average, a higher number of lines of code than the rest of the functions. When the number of lines of code grow it can make functions more complex and harder to fully test them. Therefore, it is possible that the number of lines is related with the tendency to contain security vulnerabilities, in which functions with a higher number of lines tend to be more prone to contain security vulnerabilities.

Schroeder [61] also found a positive correlation between cyclomatic complexity and the number of defects, as functions and methods that have higher complexities tend to also contain the most defects. And, since occasionally the program size is not a controllable feature of software, McCabes's number has been used as a guideline to reduce the number of faults in software. Therefore, we made an analysis to understand if the cyclomatic complexity is also a good guideline of functions with security vulnerabilities. So we compared the complexity of the faulty functions in each software project with the rest of the code. The results are presented in Figure 3.5. The graph presents the probability of the presence of a software fault as function of cyclomatic complexity.

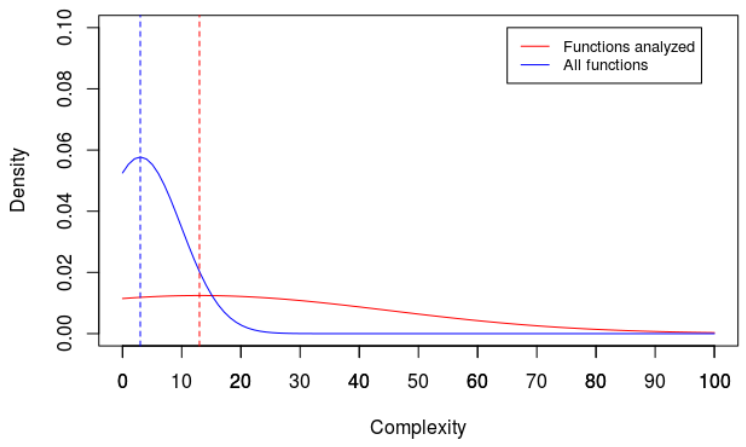

Figure 3.5: Complexity distribution of the functions analyzed and all functions

As it is possible to observe in Figure 3.5, the functions analyzed tend to have, in average, a higher complexity than the rest. When the cyclomatic complexity of the functions tend to grow it can become harder to fully test them. Therefore, it is possible that the cyclomatic complexity is related with the tendency to contain security vulnerabilities. It is also possible that a higher complexity is related with a higher number of lines of code. Even though it is not a direct relation, functions with a low number of lines of code are unlikely to have high cyclomatic complexities. Thus, we can conclude than functions with higher complexities may be more prone to contain security related faults.

As we can see both metrics can be a good indicator of problematic functions, although these two metrics are not totally independent. However, this information alone cannot fully prove that the functions with higher complexity are more prone to have faults that cause security related problems. It is needed a more detailed research to validate these findings and to derive more generalized results. However, for the experiment we are trying to produce, these results already provide us with enough information for what we want to accomplish. Therefore, we used this metric as a reference to find similar functions to conduct our practical approach. Once again, we analyzed the functions referred in the data set to understand how their cyclomatic complexity is distributed. The results are presented in Table 3.3.

Table 3.3: Cyclomatic complexity of functions with software vulnerabilities

		Cyclomatic complexity				
		Minimum	First quartile	Median	Third quartile	Maximum
Software	Linux	1	6	12	25	187
	OpenSSH	1	9	16	31	85
	Xen	1	4	13	43	927
Average		1	6.3	13.7	33	388.7

The results presented in Table 3.3 are very important for the project because they justify that the operators presented in Table3.1 could also be applied to our project. In order to apply the most frequent operators, we need to use functions that are similar to the ones that contain security vulnerabilities. As it is possible to observe the smaller complexities are very similar, however in the higher complexities such as the third quartile and the maximum the Xen presents a much higher complexities compared with the others.

We calculated the median of the values to avoid the outliers that could be found in the data set. If we used the mean, this would not represent how the values are distributed. And, as we can observe, the maximum complexity of Xen's functions is very high and it could become difficult to find functions with similar complexities to the average.

Another observation that was made during the expansion of the analysis was that many vulnerabilities seem to be related with the constants in the code. Some examples of real vulnerabilities taken from the three software projects analyzed that are related with constants are shown bellow.

```
        if (!(quirks & CP_RDESC_SWAPPED_MIN_MAX))
                return rdesc;

+       if (*rsize < 4)
+               return rdesc;
+
        for (i = 0; i < *rsize - 4; i++)
                if (rdesc[i] == 0x29 && rdesc[i + 2] == 0x19) {
                    rdesc[i] = 0x19;
```

Listing 3.1: Example of a software vulnerability 1

In example 3.1 it is possible to observe that to correct the software vulnerability it was required the addiction of an *if* construct plus statements. In the binary expression used as condition it was used a constant in order to terminate the function earlier and prevent a denial of service through integer underflow. Therefore, if the function was tested where the value of rsize was higher and lower than four it would be possible to expose the fault.

```
        for (i = 0; i < DRM_VMW_MAX_SURFACE_FACES; ++i)
                num_sizes += req->mip_levels[i];

-       if (num_sizes > DRM_VMW_MAX_SURFACE_FACES *
-           DRM_VMW_MAX_MIP_LEVELS)
+       if (num_sizes <= 0 ||
+           num_sizes > DRM_VMW_MAX_SURFACE_FACES *
DRM_VMW_MAX_MIP_LEVELS)
                return -EINVAL;

        size = vmw_user_surface_size + 128 +
```

Listing 3.2: Example of a software vulnerability 2

To correct the vulnerability shown in 3.2 it was required to add a binary expressions (num_sizes <= 0). Once again on one of sides of the expression is a constant. Therefore, if this condition was tested the num_sizes with a value above and another value below zero, the bug would probably arise and be corrected.

```
if (options->permit_local_command == -1)
                options->permit_local_command = 0;
-       if (options->use_roaming == -1)
-               options->use_roaming = 1;
+       options->use_roaming = 0;
        if (options->visual_host_key == -1)
                options->visual_host_key = 0;
```

Listing 3.3: Example of a software vulnerability 3

In the example shown in 3.3 it is possible to observe that the constant attributed to the options-¿use_roaming was wrong. It was also used another constant in the binary expression used as condition for the *if* statement. Therefore, if the function was testing where options-¿use_roaming had the value -1 and a value different that -1 would probably result in the detection of the bug and posterior correction, preventing the occurrence of a security vulnerability.

```
+
+       if (nresp > 100)
+               fatal("%s: too many replies", __func__);
+
        for (i = 0; i < nresp; i++) {
                int j = context_pam2.prompts[i];
```

Listing 3.4: Example of a software vulnerability 4

In order to correct the vulnerability shown in 3.4 it was added a *if* construct and statement. The binary expression used as condition for the *if* construct uses a constant in order to prevent the occurrence of an integer overflow and posterior privilege escalation. In this case, it was required to test the function with a value higher than 100 in order to detect the vulnerability, but once again the vulnerability shows to be related with constants.

```
if ( op_bytes == 8 )
+         {
               vcpu_must_have_cx16();
-         op_bytes *= 2;
+              op_bytes = 16;
+         }
+         else
+              op_bytes = 8;
```

Listing 3.5: Example of a software vulnerability 5

In example 3.5 the problem was that the value of op_bytes was being miscalculated. There was missing an *else* construct and statement with an attribution of a constant to op_bytes. In this case the function needed to be tested where op_bytes had a value equal to eight and a value different than eight, and in the second case, the bug would probably arise.

As it is possible to observe many of the vulnerabilities analyzed seems to be related with the constants present in the code. Even though they may not affect directly the vulnerability, as they are only used to calculate the value that causes the problem, but in many cases they appear related with if-statements that cause the vulnerabilities.

This page is intentionally left blank.

Chapter 4

Testing for security vulnerabilities

With the results collected by Barbosa et al. [7] and the results obtained from our analysis we can make the following considerations.

- The majority of the vulnerabilities only affects one function and up to three operators.
- Functions with higher complexities tend to be more prone to contain security vulnerabilities.
- The most frequent faults associated with security vulnerabilities are related with function calls and if-statements.
- The analysis of real security vulnerabilities showed that some were related with the constants present in the code.

4.1 Workflow and implementation

We decided to explore the results obtained from our analysis and create an approach to meet the vulnerabilities analyzed. Firstly, we created a tool that designs test cases based on the use of the constants present on the code. This methodology represents a grey-box testing in which the code is tested as a black-box, however, we take advantage of some features of the code to create the test cases. We also decided to focus our approach on the detection of extraneous faults. As the extraneous type of faults do not change the output of the software, we explored that and developed a methodology to detect the statements that do not alter the outcome of a program.

The workflow of the tool is presented in Figure 4.1.

The tool was developed in Java and uses the CDT plugin from Eclipse to parse the source code and build the respective Abstract syntax tree (AST). We chose to use the CDT plugin from Eclipse to help in the implementation since it provides very useful characteristics to support source code modifications. This plugin also uses the Visitor Pattern which means that it traverses all the nodes from the AST. So for the literal based testing, the tool inspects all the literals found in the AST. Based on the formal parameters of the target function, the constants are either used or throw away. For example, if a functions receives a integer as input, all the integers are saved while the other data types get discarded. Although some of them could be casted, we decided to not do this because

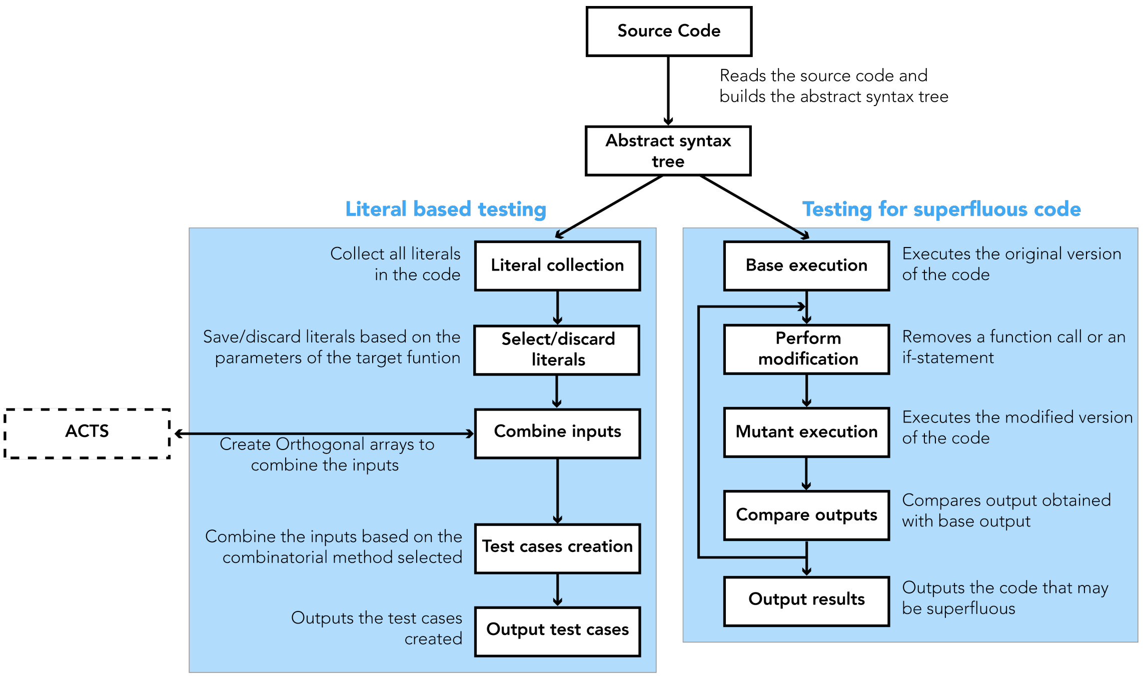
Source Code
Reads the source code and builds the abstract syntax tree
Abstract syntax tree
Literal based testing
Testing for superfluous code
Collect all literals in the code
Literal collection
Save/discard literals based on the parameters of the target funtion
Select/discard literals
ACTS
Create Orthogonal arrays to combine the inputs
Combine inputs
Combine the inputs based on the combinatorial method selected
Test cases creation
Outputs the test cases created
Output test cases
Base execution
Executes the original version of the code
Perform modification
Removes a function call or an if-statement
Mutant execution
Executes the modified version of the code
Compare outputs
Compares output obtained with base output
Output results
Outputs the code that may be superfluous

Figure 4.1: Developed tool workflow

this way we would not use the literal found but a modified version of it. In addition, if the target function does not have any formal parameters, the code is analyzed in order to perceive whether or not the inputs are received by standard input, i.e. received from the terminal/console. If it's the case, the tool generates text files based on the strings found in the source code and some standard files. These standard files are composed by an empty file, a small file and a big file.

The developed tool also have the option to create test cases using a combinatorial test case generation. When the AST has been traversed and all the possible inputs saved, it uses the functionalities of the Automated Combinatorial Testing for Software (ACTS) tool [62]. This tool receives the list of formal parameters and a list of inputs for each parameter and generates a combinations of inputs in order to increase the test coverage without exponentially increasing the number of test cases.

At first, we also wanted to use the limits of each data type in the generated test cases. This way, we could also test for boundary faults. However, when applied to the selected function, they would crash. After some research we discovered that the functions were crashing because they were allocating too much memory. Some of these functions use dynamic memory allocation in which the size depends on the parameters received, therefore, when the function tries to allocate a enormous quantity of memory it overflows the stack. So we had to abandon this idea and continue only with the constants present in the code.

We also created a more focused approach aimed at the extraneous type of faults. In real software projects the correction of this type of faults is done by the removal of instructions. Thus, if the software continues to operate properly, it means that these instructions may not be contributing to the system's output as the system continues to operate correctly. Therefore, they may be useless to the program and cause security problems as they may carry external bugs.

To this extent, the main focus of our approach was the removal of small parts of the code, more specifically, function calls and if-statements, and evaluate whether or not the program would still be executed properly. We used function calls and if statements because, as observed in the previous section, the most frequent faults related with security vulnerabilities are associated with function calls and if-statements. The main idea behind our approach is to create a base output, from the normal execution of the program. Then remove a small part of the code, execute the program again and compare the output given with the base output. If the output is the same, the removed part of the code may be unnecessary. After this, repeat the same process for every function call and if-statement.

This approach was integrated in the previously developed tool. It also uses the CDT plugin to perform the modifications to the AST. As the objective was to remove small parts of the code, the easiest way to do it is to emulate missing type faults. These faults correspond to the emulation of the MFC and the MIFS operators. Therefore, we re-used part of the ucXception [40] and applied it to the problem. The tool creates the AST based on the target function, and when certain constrains are met, it performs the necessary modifications to the AST. When a mutant version of the program is created, the tool compiles it, executes it and saves the output given. This output is then compared with the output from the initial version of the program. In order to compare the outputs, the Jaro Winkler distance between them is calculated. The Jaro Winkler distance is a metric used to calculate the distance between two strings that outputs a number between 0 and 1. Based on the number obtained, the tool is capable of distinguishing the similar executions and warn about the part of the code that may be unnecessary.

4.2 Usage

In order to be able to use the developed tool, it is important to note some particularities related to its usage. The structure of the command to run it is the following.

```
java -jar TestingTool.jar File [Option...]
```

File: .c or .cpp file

Option:

- **-p - Pair wise option** - Uses the ACTS tool to create orthogonal arrays to combine the inputs created.
- **-c - C Parser** - Uses the GNU C Parser as provided by Eclipse CDT. (default)
- **-cpp - C++ Parser** - Uses the GNU C++ Parser as provided by Eclipse CDT.
- **-f - Focused testing [Function...]** - Experimental method that removes functionalities and compares the output in order to find extraneous code. (The function name is case-insensitive)
- **-k - Keep output files** - Keep the auxiliary files used during the execution.
- **-h - Help** - Show the help menu.
- **-d - Debug option** - Prints the operations the tool is performing and additional notes related with the execution.

Below are presented some examples of commands to execute the testing tool, for different situations.

```
$ java -jar TestingTool.jar file.c -p
```

This command creates test cases using orthogonal arrays to combine the inputs, for the functions present in file file.c.

```
$ java -jar TestingTool.jar file.c -d -f
```

This execution activate the DEBUG option which shows information related to the steps that are made by the tool, and it uses the approach focused on the extraneous type of faults for all the functions present in file file.c.

```
$ java -jar TestingTool.jar file.c -f=function
```

This execution differs from the previous one by specifying the function where the focused testing should be perform.

4.3 Limitations

The developed tool is capable of creating test cases and testing a vast set of function, however, it contains some limitations that restrict the extension of its use.

One of these limitations is when the target function uses preprocessor directives and macros. The tool was developed using the features of CDT plugin and in order to build the AST the plugin parses the source code, removing certain parts of the code. Thus, when the tool searches for extraneous constructs creating mutant versions of the original program, the mutant version does not contain these directives and macros. One solution to this problem could be an initial scanning of the code to save all the statements that the CDT plugin discarded. In addition, when the mutant versions of the program were created, the tool would insert the saved statements again.

Another limitation is the data types that it covers. The developed tool is capable of creating test cases for functions that receive standard C data types as formal parameters and for functions that receive inputs by terminal/console. However, there is a much larger set of data types available that the tool does not cover. Another problem concerns the function that receives arrays and structures as input. Even though it is not a limitation, since the tool is capable of creating inputs for these cases, they usually present worst results than the functions that receive simple data types. However, the tool is limited to arrays of simple data types and to only one dimension.

Chapter 5

Results

In order to evaluate the feasibility of the developed tool, it was decided to perform a representative security vulnerabilities injection to compare its detection of security vulnerabilities with the commonly used testing techniques. With this in mind, in this chapter, we present the results from the application of two testing techniques in comparison with the results obtained from the developed methodology.

First, we searched for functions publicly available, which followed approximately the complexities presented in Table 3.3. As said before, it is important to use functions that follow similar complexities with the functions analyzed in order to be able to apply similar faults. In other words, in order to inject software vulnerabilities it is required software that is prone to contain security vulnerabilities. Most of the functions were picked from The Computer Language Benchmarks Game web page [63] and one was provided by the advisor of this dissertation. We decided to use these functions because the objective of the web page is to compare the performances of simple algorithmic problems between different programming languages. Therefore, it contains the best implementations of the problems for each programming language. In addition, we also needed to use functions without faults, however, we cannot prove that some piece of code is free of faults. Thus, using the best implementations we could reduce the probability of the code already containing faults.

To perform the fault injection, we decided to focus on the five functions with a small complexity and five functions with a medium complexity, according to the results presented in Table 3.3. We chose to not include functions with high complexity due to the difficulty of finding functions with similar complexity. Although it still represents a large set, it is important to consider that the functions analyzed have a complexity higher than the usual. Hence, the difficulty in finding similar functions.

The description and some metric of the selected functions are presented in Table 5.1.

For the representative emulation of security vulnerabilities, we decided to focus on two missing, two wrong and two extraneous operators. Based on the most frequent fault operators, we decided to focus on the EFC and the EIFS operator for the extraneous construct, the MIFS and the MFC operator for the missing construct and for the wrong construct, we choose the WLEC and the WALR operator.

To inject the security vulnerabilities, we used a software fault injection tool previously studied, the ucXception suite [40]. This tool produces a set of patches, each consisting of

[1]Perfect binary tree - A binary tree in which all internal nodes have two children and all leaves are at the same level.

Table 5.1: Functions selected for the fault injection

Function	Description	Cyclomatic complexity	Number of lines
binary-trees	Create perfect binary trees [1]	4	47
pidigits	Generate digits of Pi	5	22
hondt	D'Hondt method implementation	6	38
regex-redux	Manipulate DNA sequences	6	77
k-nucleotide	Count all the nucleotide sequences	7	21
n-body	Model the orbits of Jovian planets	12	39
fasta	Generate DNA sequences	12	58
fannkuch-redux	Fannkuch benchmark implementation	13	56
reverse-complement2	Converts a DNA sequence into its reverse	14	46
reverse-complement	Converts a DNA sequence into its reverse	14	70

a single emulation operator instance. However, as this tool is targeted at the emulation of software faults, it does not feature the operators required for the emulation of security vulnerabilities, such as the WALR, the EFC and the EIFS. The WALR was based on one operator that the tool supports, the Missing small and localized part of the algorithm (MLPA). The MLPA operator consists on the removal of a small part of the code. Thus, to create the desired fault, the WALR, we changed the patch file to, instead of removing a small part of the code, place that part in a different location. The rest of the operators were manually created based on the real defects analyzed in the data set in order to be the more realistic possible. The number of patch files created for each function is presented in Table 5.2.

Table 5.2: Number of patches for each function

Function	Missing		Wrong		Extraneous		Total of
	MIFS	MFC	WLEC	WALR	EIFS	EFC	patches
binary-trees	0	5	5	4	3	5	22
pidigits	3	9	4	4	1	4	25
hondt	1	0	5	2	1	0	9
regex-redux	1	15	6	3	3	2	30
k-nucleotide	2	8	12	5	3	4	34
n-body	0	1	9	4	4	3	21
fasta	4	1	8	3	3	5	24
fannkuch-redux	2	1	8	3	3	2	19
reverse-complement2	6	3	9	6	4	2	30
reverse-complement	4	10	7	8	3	4	36
Total	23	53	73	42	28	31	250

As it is possible to observe in Table 5.2, there are, in some cases, zero patches for an operator. This can happen if the function in question does not contain the instructions that the operator affects. For example, the n-body function does not contain if conditions, therefore it is impossible to emulate the MIFS operator. The same problem occurs in the hondt, where the function does not call any external function.

As it is also possible to observe, the number of patches from the missing and extraneous type are relatively close, however, the wrong instances have a much higher number. This can happen because the missing constructs have the limitations of needing the instruction that the operators affect. The EFC has the limitation as it is required to have some external functions to call in order to implement such faults. As in the case of the hondt function, it does not have any function, therefore, the number of EFC operators are zero. However, the wrong construct does not have any limitations, and its nature allows for large growth of faults. For example, in a *if* statement, if the condition is `i < 10`, applying the WLEC operator, we can change the condition to `i <= 10`, `i > 10`, `i >= 10`, `i == 10` or `i != 10`, and all of them represent a wrong logical expression.

First, we decided to evaluate the detection of a black-box testing technique. We decided to use random testing as it is commonly used in the industry since it represents a technique that it is cheap to implement and easy to understand [64], and then complemented with a more sophisticated technique. It is also able to detect the basic faults that can be found by simply executing the program.

The metric used to evaluate the effectiveness of the testing technique is based on the mutation score in the domain of mutation testing. However, in our analysis we cannot refer to mutation score since it is a representative software fault injection. What we are trying to accomplish is an emulation of real mistakes that programmers make, instead of simple mutations on the source code. Therefore, we define it as detection rate, which is the quotient between the number of detected faults and the total number of injected faults.

The testing environment was created by an oracle, whose objective was to compare the output given by the correct version of the program with the output from the program with the fault. Since we are performing random testing, the function needs to receive randomly generated inputs. These inputs are created using a pseudorandom number generator and adapted to the domain of each function. However, we needed to initialize the generator with the same seed in both executions (the correct and the faulty one) to prevent it from creating different inputs in executions that must be similar. Each function is executed several times in which the number of executions depends on the complexity of the functions. In order to simplify the method, we defined the number of executions as the McCabe's number of the functions. Therefore, the more complex functions are executed more times than the smaller ones.

We applied several faults using the respective patch files, which contained the modification necessary to emulate a security vulnerability. Then we compared the output from the faulty execution and the correct output. If they are equal, it means that the injected fault was not detected. On the other hand, if the output is different then it means that fault was detected by the testing technique. After testing all the functions, we obtained these results presented in Table 5.3.

It is possible to observe in Table 5.3 that, on average, almost 60% of the emulated faults remain undetected using a random testing technique. As we can see, in some of the functions no fault is detected. This can happen because some functions need to receive as input a string in the FASTA format. The FASTA format is a text based format used to represent nucleotide sequences or peptide sequences [65]. Therefor, when the input is a random generated string the program stops the execution earlier because the input given is not in the correct format. Thus, the program is not completely exercised, not allowing the detection of the injected faults.

As we are injecting faults that represent security vulnerabilities, the ones that are being overlooked can be the ones that cause the security vulnerabilities. In terms of security, detecting 40% of all faults is a low detection ratio. In a real software project it is unthinkable to deploy it with this amount of security vulnerabilities still present in the source code. Therefore, one can understand that, even though we are applying a shallow

Table 5.3: Results from random testing

Function	Number of faults	Test suite size	Undetected faults (%)
binary-trees	22	4	54.6
pidigits	25	5	32.0
hondt	9	6	55.6
regex-redux	30	6	46.7
k-nucleotide	34	7	100.0
n-body	21	12	33.3
fasta	24	12	41.7
fannkuch-redux	19	13	31.6
reverse-complement2	30	14	96.7
reverse-complement	36	14	100.0
Total	250	93	60.4

technique, there are improvements that can be made.

To obtain a better comprehension of the types of faults that are being overlooked, our results were grouped according to their nature (missing, wrong and extraneous). The results obtained are presented in Table 5.4.

Table 5.4: Detailed results from random testing

Function	Missing		Wrong		Extraneous	
	Total	Undetected	Total	Undetected	Total	Undetected
binary-trees	5	2	9	4	8	6
pidigits	12	3	8	3	5	2
hondt	1	0	7	4	1	1
regex-redux	16	10	9	3	5	1
k-nucleotide	10	10	17	17	7	7
n-body	1	0	13	6	7	1
fasta	5	2	11	3	8	5
fannkuch-redux	3	0	11	4	5	2
reverse-complement2	9	8	15	15	6	6
reverse-complement	14	14	15	15	7	7
Average		64.5 %		64.3 %		64.4 %

Random testing was unable to accomplish a higher detection rate because it is a shallow technique that does not take into consideration the structure of the code. As the inputs are randomly generated, the technique could be always exercising the same execution path, missing the execution paths that would traverse through the inserted fault.

As mentioned before, some functions do not detect any type of faults due to the inputs being randomly created. However, in the *reverse-complement2* function, one fault is detected. This happens because one of the faults is injected before the verification of the input, making it possible to detect that fault.

Also as we can observe in Table 5.4, there is no specific type of fault that requires more attention since random testing detects all types of faults likewise since all types of faults have a similar detection rate. As said before, random testing is a shallow technique from which it is expected to detect only the basic faults.

As said before, random testing is usually complemented with a more sophisticated technique. We decided to evaluate control flow testing because it represents a more focused technique since it is based on an analysis of the internal structure of the code. Moreover, most of the bugs result in control flow errors, therefore, they can be caught using control flow testing [47]. In order to test the selected functions, we created a set of test suites for each function. It is important to note that we created test cases to test the faulty version of the program. This means that if the fault injected creates a new execution path, this path is also tested. Also if a fault removes a execution path, the test case that tested that path is removed. When testing real software, the testers create the test cases based on the code with bugs, therefore we need to follow the same approach.

The testing environment is similar to one created for the random testing. One of the differences is that each program is executed with inputs manually created based on the control-flow of each program. Then for each fault, inputs are added or removed according to the type of fault. Then the oracle compares the output given by the original version of the program with the output from the program with the fault. If the outputs are different, it means that the fault was detected by the test suite. If the output from the mutant program is equal to the output given by the correct version of the program, then the type of fault is collected. After testing all the functions we obtained the results presented in Table 5.5.

Table 5.5: Results from control flow testing

Function	Number of faults	Test suite size	Undetected faults (%)
binary-trees	22	2	54.6
pidigits	28	5	28.0
hondt	9	6	11.1
regex-redux	30	3	50.0
k-nucleotide	34	3	32.3
n-body	21	2	38.1
fasta	25	5	41.7
fannkuch-redux	19	3	36.8
reverse-complement2	30	2	56.7
reverse-complement	37	5	47.2
Total	250	36	42.0

Comparing these results with the ones obtained using random testing we can observe that the detection rate has increased slightly. This behavior is expected since we are using a more thorough testing technique, allowing it to exercise more thoroughly all the execution paths functions. Also, unlike what happened with random testing, all the functions have faults that have been detected. As control-flow has access to the source code and its specification, we could create test cases in the required format for each function.

Although the detection rate is higher that in random testing, it still fails to detect many vulnerabilities. The requirements for security testing are more restricted, therefore, detecting 60% of all injected faults is still a low detection rate. Thus, there are some improvements that can be made in order to detect those vulnerabilities.

To get a better comprehension of the types of faults that are being overlooked, our results were grouped according to their nature (missing, wrong and extraneous). The results obtained are presented in Table 5.6.

As one can observe in Table 5.6, the undetection rate is near 40%, except for the

Table 5.6: Detailed results from control flow testing

Function	Missing		Wrong		Extraneous	
	Total	Undetected	Total	Undetected	Total	Undetected
binary-trees	5	2	9	4	8	6
pidigits	12	2	8	3	5	2
hondt	1	0	7	1	1	0
regex-redux	16	10	9	3	5	2
k-nucleotide	10	1	17	7	7	3
n-body	1	0	13	4	7	4
fasta	5	2	11	3	8	5
fannkuch-redux	3	0	11	5	5	2
reverse-complement2	9	4	15	8	6	5
reverse-complement	14	6	15	8	7	3
Average		39.5 %		34.8 %		59.4 %

extraneous type of faults. Control-flow represents a more complete testing technique, as it tests all the execution paths of the functions, therefore, it is expected to present better results than random testing.

The extraneous type of faults are the ones less detected in our experiment and the ones that appear often associated with security faults, making us conclude that these types of faults are the ones that need more attention in software development. As the instructions do not affect the output of the functions, the commonly used testing techniques may fail to detect this type of defects.

The detection rate of wrong operators is higher due to the type of fault it creates. Altering the structure of the code by changing the position of a block of instructions or changing the condition of a branch instruction can alter the whole execution of the code. This way, the testing technique can easily detect the presence of a fault. However, the detection rate is not higher because from the change of place of an instruction block may also result no change. A clear example of this is when the instructions are independent, i.e. if we have two variables and need to make an operation to each one, and neither the operations nor the variables have a relation with each other, it is normal to change the order of the instructions, as there is no change in the output.

The missing constructs have a similar detection rate than the wrong. As we are using a technique focused on exercising all the execution paths, it was able to exercise the execution paths that would pass through the removed instructions, detecting the ones that are missing. In contrast to what happened with random testing, as inputs are randomly generated they may be following the same path and not fully exercising the program.

Using the developed tool, we were able to create test cases for each selected function. We developed a oracle similar to the ones used previously. It executes the program with the test cases created by the developed tool to get the correct version of the output. Then it injects a fault into the program and executes it again with the same inputs and compares the output. If the output is equal, it means that the fault was not detected. After testing all the functions, we obtained the results presented in table 5.7.

In general, we can see that our methodology presents a higher undetection rate than the two testing techniques previously evaluated. However, with the missing type of faults we were able to get better results than random testing.

Another problem we found was that the test cases involving structures, arrays and

Table 5.7: Detailed results from literal based testing

Function	Missing		Wrong		Extraneous	
	Total	Undetected	Total	Undetected	Total	Undetected
binary-trees	5	2	9	3	8	5
pidigits	12	3	8	3	5	2
hondt	1	1	7	5	1	1
regex-redux	16	9	9	3	5	1
k-nucleotide	10	10	17	17	7	7
n-body	1	1	13	13	7	7
fasta	5	5	11	11	8	8
fannkuch-redux	3	0	11	6	5	3
reverse-complement2	9	6	15	13	6	6
reverse-complement	14	6	15	15	7	7
Average		56.6 %		77.4 %		79.7 %

strings have worse results than the rest. The developed tool is capable of creating test cases involving structures, however, as it is a more complex data type, the idea of using the constants present in the code may not be enough. The same happens with the functions that receive inputs via terminal/console. The tool is capable of detecting if the functions use the *stdin* to receive the inputs, and creates a set of text files to exercise the target function. However, using only the strings present in the code may be a shallow technique and it is necessary to have a deeper understanding of the function to perceive the type of inputs it receives. One of the cases is when the functions need to receive as input a string in the FASTA format. First the function analyze the input received in order to understand if it presents the correct format. However, as the test cases are created based on the strings present in the code, the tool is not capable of understanding of the inputs specifications, presenting worse results than the functions that receive the simple data types as input.

As one can observe, in some cases there were functions in which no fault was detected. This problem arises the same way as in random testing. Although the test cases were created based on the constants present in the code, the functions are still tested as a black-box. Therefore, the developed methodology encountered the same problem with the functions that need to receive as input a string in the FASTA format. As the input do not correspond to the required format the functions stop the execution earlier, preventing the complete exercise of the code.

However, we were expecting to obtain a slightly higher fault detection rate than random testing. Since our approach tests the code similarly to random testing but in a more focused way, we were expecting it to be able of producing better test cases. However, the main objective of this dissertation is to complement the commonly used software testing techniques. Thus, we made an analysis of the faults not detected by the used testing techniques (random and control flow) in comparison with the faults detected by the developed approach. In table 5.8 are presented the faults that our approach detected that the previously used techniques were unable to detect.

As it is possible to see, the results show that our approach was capable of detecting some faults that other testing technique failed to do so. As said before, the objective of this dissertation is to complement the testing techniques, and for the selected functions we were able to accomplish it. The developed tool detected two vulnerabilities in the *binary-trees* function. This function only receives one input which is a integer, therefore,

Table 5.8: Faults detected only by our methodology

Function	Missing		Wrong		Extraneous	
	Total	Undetected	Total	Undetected	Total	Undetected
binary-trees	5	0	9	1	8	1
pidigits	12	0	8	0	5	0
hondt	1	0	7	0	1	0
regex-redux	16	0	9	0	5	0
k-nucleotide	10	0	17	0	7	0
n-body	1	0	13	0	7	0
fasta	5	0	11	0	8	0
fannkuch-redux	3	0	11	0	5	0
reverse-complement2	9	0	15	0	6	0
reverse-complement	14	0	15	0	7	0
Average		0.0 %		0.9 %		1.7 %

as the tool uses the constants present in the code it may have been able to find a corner case that the other testing techniques did not cover.

Our tool was also used to detect the presence of functionalities that may be unnecessary to the correct operation of the software. We developed an oracle that injects extraneous faults and tests the functions with the developed tool. The tool executes the original program to create a base output. Then it removes a functionality and executes the program again and warns if the removed functionality was superfluous or not, i.e. if the removed functionally affected or not the outcome of the program. If the tool warns that the extraneous fault injected is unnecessary, it means that the fault was detected. If the fault injected alter the output, then the tool does not consider it superfluous, which means that the fault was not detected. The results obtained are presented in Table 5.9.

Table 5.9: Focused extra functionalities

Function	EIFS		EFC	
	Total	Undetected	Total	Undetected
binary-trees	3	1	5	1
pidigits	1	0	4	0
hondt	1	0	0	0
regex-redux		—		—
k-nucleotide	3	1	4	2
n-body	4	1	3	0
fasta	3	2	5	2
fannkuch-redux	3	1	2	0
reverse-complement2	4	1	2	1
reverse-complement	3	1	4	2
Average		32.0 %		27.6 %

As is it possible to observe, the undetected faults represent 30% of all the faults injected. Comparing these results with the ones previously obtained from the control flow and random testing, this methodology seems to be able to reduce the undetection rate from 60% to 30%, which represents half of the undetection rate of commonly used testing techniques.

In one of the selected function, we were unable to extract results. The regex-redux function uses C preprocessor directives. And when the CDT plugin parses the code, it removes all the directives and comments present in the code. Thus, when the tool creates the mutant versions of this program, they do not contain the directories, which makes it unable to execute. This way, it was not possible to extract results from it.

The EIFS operators represent a bigger difficulty to be detected using our approach because these operators can represent a larger number of instructions. In other words, a function call is usually composed with only one instruction, however, the if-statement is usually composed by a varied number of instructions, always larger than one. Therefore, the removal of an if-statement is more prone to change the output of the program. And, in this case, if the output is different than the base output, it means that the fault was not detected.

Although some of faults remain undetected, and in terms of security this situation is still a matter of concern, we were able to complement the testing technique studied, which is the main objective of this dissertation.

This page is intentionally left blank.

Chapter 6

Discussion

6.1 Advantages and disadvantages

The main advantage of our approach is that it is capable of detecting security vulnerabilities that the commonly used testing techniques fail to do. Based on the results from the previous section, it was possible to observe that our approach is capable of detecting two security vulnerabilities that random testing and control-flow testing are unable to detect. In addition, regarding the extra operators, our methodology was able to increase the detection rate of the other testing techniques by 30%, which is the double what they had achieved.

A disadvantage of the tool is that it still has some limitations, as it cannot be applied to all types of functions. Before using it, it is important to keep in mind the limitations the tool presents. As said before, the detection of security vulnerabilities in functions that receive arrays and structures presented worst results than in functions that receive standard C data types. In addition, for the superfluous code testing, when the target function should not contain C preprocessor directives as the CDT plugin removes them making it impossible to produce results.

Another disadvantage is that the developed tool alone does not have a high detection rate. The objective was to complement the other testing techniques and not the development of a new methodology capable of detecting all security vulnerabilities. Therefore, it is recommended to use other testing techniques to test the software and then complement them with the developed tool.

6.2 Work plan

Initially, the work plan was defined so that all the objectives of this project could be completed. With this draft, it was possible to plan ahead the allocation of time needed for each task.

First Semester

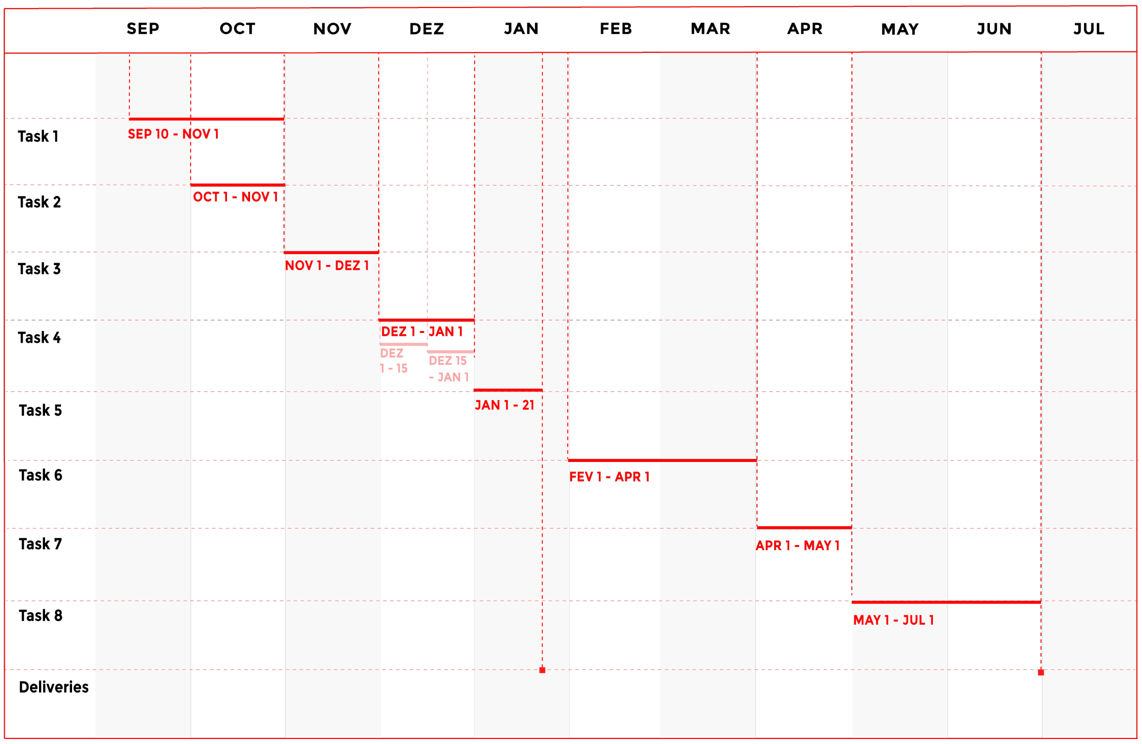

Figure 6.1: Gantt chart for the expected work plan

Task 1 - State of the art study

The first task consisted in developing the knowledge of the topics on which this project is based, with particular attention to software testing, emulation of software faults, security vulnerabilities and software faults in general. This work should result in the writing of the state of the art chapter in the dissertation.

Task 2 - Study of security faults

It is important to understand the conditions that lead a simple mistake to create a security vulnerability. These conditions can be related to the fault itself or the code in which the fault is found. Therefore, it is crucial to fully understand the security faults to proceed to the next stages. This study was already initiated by us, but for this dissertation it was decided to deepen the analysis to obtain more detailed results.

Task 3 - Choice of tools and preparation of the test environment

This task consists in choosing the best tools for the test suites creation based on the test methods previously studied. It will also be necessary to choose a representative set of functions and fragments of code from open-source projects in C language that will be targeted on the test suites. The ucXception tool will be used for the software fault emulation, followed by an approach inspired by mutation testing.

Task 4 - Evaluation of the existing test methodologies

The objective of this task is to evaluate the software testing methodologies chosen during the previous studies (e.g., control-flow testing, data-flow testing and black-box testing). An emulation of faults in various functions of the chosen code will be performed, using a tool already created (ucXception). The defects to be inserted will be representative of security vulnerabilities, being this is the fundamental point of this task. The evaluation consists in counting the faults that the tests methodologies cannot detect and determine how often this behavior happens. This task is divided into two smaller sub tasks.

Task 5 - Writing the intermediate report

The tasks made within the first semester must be documented in the form of an intermediate report, followed by the public presentation and discussion. In this disseration, it is important to present the results of the evaluation of the existing software testing methodologies, identified in the state of the art.

Second Semester

Task 6 - Development of the practical approaches

With the results of the evaluation made previously, it was possible to identify the most frequent faults that can go unnoticed. Therefore, the objective of this task is to develop an approach to complement the software testing methodologies studied.

Task 7 - Evaluation of the developed approach

This task consists in the evaluation of the approach proposed in the previous phase. The objective is to compare these results with the results obtained using the common methodologies. With this it is possible to conclude whether or not we can to complement the software testing techniques with a more focused approach.

Task 8 - Writing the Master's dissertation

The purpose of this task is to complete the writing of the master's thesis. It also includes the feedback from the advisors and their respective indications. The final report must document all the work done in the scope of software verification aimed at security vulnerabilities.

6.3 Work plan alterations

Some of the tasks did not meet the initial work plan. In the first semester the preparation of the testing environment took more time than the expected. The initial idea was to work with code taken from open source projects but soon it was possible to understand that this could represent a laborious task. As the intention was to focus on simple functions, it was necessary to have code with few or no dependencies, either from code or from libraries to reduce the possibility of external faults. Therefore, it was essential to make a more extensive research to find more adequate functions for the project. The third task was also delayed due to difficulties on finding tools to automate the program testing. Even though some tools were found, they did not work with the selected functions. The evaluation of the existing test methodologies was also delayed because there was the expectation to find a more helpful tool. Because of this delay, it was decided to split the evaluation of the testing techniques into the two semesters, leaving one testing technique for the first semester and another for the second.

In the second semester, a second research was done on tools to automatically test the selected programs but more focused on automatic white-box testing. The development of the practical approach also took more time than the expected as we decided to also explore a more focused approach based on the results obtained in the previous tasks. Therefore, task eight (evaluation of the developed approach) had to be done again in order to extract different results from the extension of our approach.

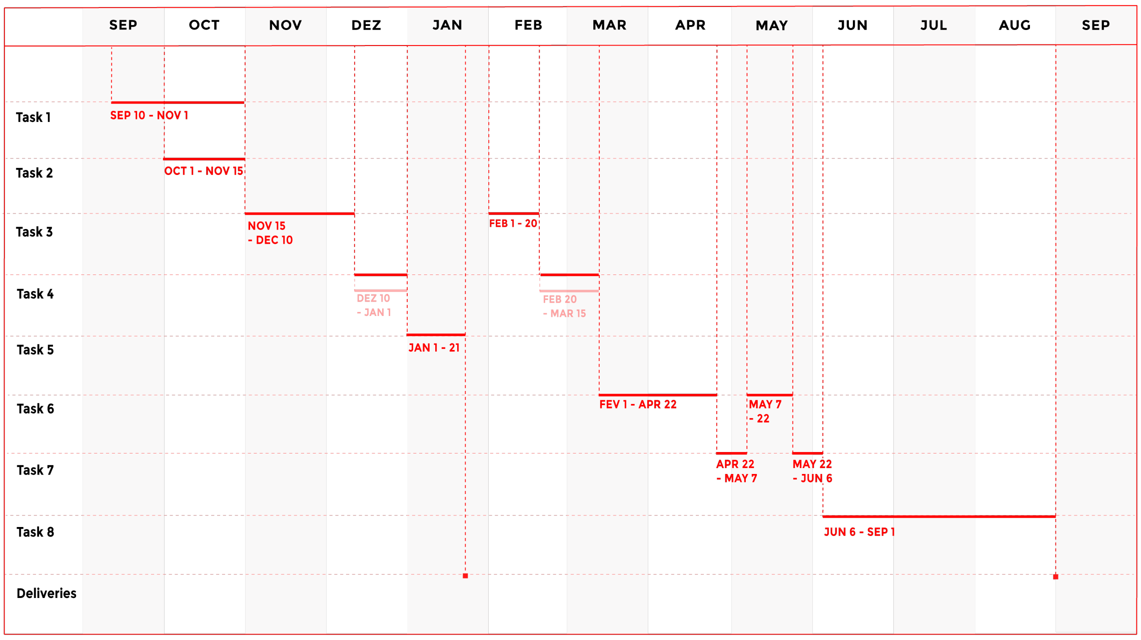

Figure 6.2: Gantt chart for the real work plan

Chapter 7

Conclusion

The main objective of this dissertation is to complement the commonly used testing techniques, considering the software faults more associated with security vulnerabilities. In order to accomplish this, we proposed four core objectives.

- Study of the conditions in which a software fault leads to a security vulnerability.
- Perform fault injection to emulate representative security vulnerabilities in order to evaluate the commonly used testing techniques and understand where they may fail.
- Develop an approach to improve upon the commonly used testing by finding more vulnerabilities.
- Evaluate the results obtained from the developed approach.

In order to complete the first objective, we took advantage of a field study previously done on security vulnerabilities and expanded its results. Based on the analysis of the security vulnerabilities, we found that vulnerabilities often seem to be related with the constants present in the code. We were also able to conclude that functions with a higher complexity are more prone to have security faults. The same happens with the functions with more lines of code. Even though there is a correlation between these two metrics, as functions with a higher number of lines tend to have a higher complexity, we concluded that they can be good indicators of the functions more prone to contain security vulnerabilities.

It was performed a representative injection of security vulnerabilities using 10 benchmark programs in order to evaluate the effectiveness of the commonly used testing techniques. The programs were selected based on their cyclomatic complexity to be similar to functions that contain security vulnerabilities. Regarding the faults injected, they were based on the type of faults that showed to be more prone to create security vulnerabilities. With the results obtained, it was also possible to understand the main limitations that the commonly used testing techniques present.

We developed an approach to complement the commonly used testing techniques. This tool was developed in Java together with the CDT plugin, which parses source code, generates the respective AST and applies modifications to it. On one hand, the developed tool is capable of creating test cases based on the literals present in the code for functions written in C. On the other hand, it provides a testing methodology focused on the extraneous type of faults. As this type of faults may not compromise the correct execution of the software, they are invisible to the common testing techniques, thus, the focus of our approach is to find code that may be superfluous.

The obtained results show that random testing is capable of detect approximately 40% of the injected vulnerabilities. Using control-flow it was possible to complement the detection rate up to 60%. Using our methodology we were not able to surpass the detection rate obtained using random testing. As the developed approach is a similar to random testing technique but creates the tests in a more focused way we were expecting to obtain better results. However, we were able to complement the other testing techniques used, as it was possible to detect two software vulnerabilities that the other testing techniques studied failed to detect.

In relation to the detection of extraneous software faults the commonly used testing techniques were capable of detecting around 40%. However, using our approach we were able to increase the detection rate to around 70%, simply looking for code that may not affect the correct operation of the program. This means that out methodology was able to double the detection rate of the other testing techniques studied in the detection of extraneous type of faults.

Based on these conclusions, future work should continue the study on software faults capable of creating security vulnerabilities, namely:

- Use a more vast type of functions in order to generalize our results.
- Apply other testing techniques to verify if the results are similar.
- Resolve the limitations of the developed tool.
- Meet the cases that the tool was unable to detect.

It is important to perform a similar fault injection in a more vast number of functions in order to understand if our results are indeed representative. Our analysis was focused on the emulation of security vulnerabilities on 10 functions which is not representative of all the software written in C. The ideal would be to use a larger number of functions and functions with different complexities. Even though our analysis covered the first quartile and median of the complexities of the functions studied, it is important to expand it to include other complexity values that we did not cover, for example functions with higher complexities.

It is also important to apply other testing technique in order to understand if they present the same limitations as the testing techniques applied in our analysis. The question being made is if we used different testing techniques, could we detect a larger number of faults? And even if the detection rate wasn't higher, were the other testing techniques able to detect the faults that random, control-flow and our approach fail to do?

Also, our tool is capable of creating a set of test cases for function that use the standard C data types, and functions that receive inputs by terminal/console. However, there is a much larger set of data types available that the tool does not cover. Also, the tool is capable of creating arrays of values and structures to use as input, however, they present worst results than the rest of the functions that receive simple data types as input. Therefore, an objective for future work is to improve the creation of these types of inputs, expand it to the types of inputs that it does not cover and resolve the other limitations that the tool presents.

There is still a large set of faults that our approach was not capable of detecting. Therefore, it is important to analyze these faults in order to understand the reason they escape. A full comprehension of these faults can be a large step towards the identification of potential vulnerabilities and posterior prevention of data and money loss on organizations. Thus, it is important to continue developing the study on faults that cause security vulnerabilities in order to understand what makes them different from the rest.

This page is intentionally left blank.

References

[1] Kshirasagar Naik and Priyadarshi Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley, 2011.

[2] Magsilva. Cause-effect graph example. Available at `https://magsilva.pro.br/apps/wiki/testing/Cause-effect{_}graph{_}example` (2019/06/11), 2009.

[3] Guru99. What is State Transition Testing? Available at `https://www.guru99.com/state-transition-testing.html` (2019/06/09).

[4] Nai-Wei Lin. Control Flow Testing. Available at `https://www.cs.ccu.edu.tw/{~}naiwei/cs5812/st4.pdf` (2019/06/11).

[5] Guru99. Path Testing & Basis Path Testing with examples. Available at `https://www.guru99.com/basis-path-testing.html` (2019/06/08).

[6] ProfessionalQA. Regression Testing. Available at `http://www.professionalqa.com/regression-testing` (2019/06/10), 2019.

[7] Raul Barbosa, Frederico Cerveira, Luís Gonçalo, and Henrique Madeira. Emulating representative software vulnerabilities using field data. *Computing*, pages 1–20, 2018.

[8] Software Testing Help. Mutation Testing: Testing Technique with a Simple Example. Available at `https://www.softwaretestinghelp.com/what-is-mutation-testing/` (2019/06/11), 2019.

[9] Tutorialspoint. What is All pairs Testing? Available at `https://www.tutorialspoint.com/software{_}testing{_}dictionary/all{_}pairs{_}testing.htm` (2019/06/15).

[10] Guru99. What is Orthogonal Array Testing (OATS)? Available at `https://www.guru99.com/orthogonal-array-testing.html` (2019/06/10).

[11] Alex Samurin. Explore the World of Gray Box Testing? Available at `http://extremesoftwaretesting.com/Articles/WorldofGrayBoxTesting.html` (2018/12/20), 2003.

[12] Testbytes. All Info About Grey Box Testing (With Examples). Available at `https://www.testbytes.net/blog/grey-box-testing/` (2019/06/15), 2019.

[13] João Durães and Henrique Madeira. Emulation of Software Faults: {A} Field Data Study and a Practical Approach. *IEEE Trans. Softw. Eng.*, 32(11):849–867, 2006.

[14] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[15] Andi Wilson, Ross Schulman, Kevin Bankston, and Trey Herr. Bugs In The System: A Primer on the Software Vulnerability Ecosystem and its Policy Implications. Technical report, New America, 2016.

[16] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 447–456. ACM, 2010.

[17] TIOBE. TIOBE Index for January 2019. Available at `https://www.tiobe.com/tiobe-index/` (2019/01/19), 2019.

[18] Menlo Park. Hackerpocalypse: A Cybercrime Revelation. Available at `https://cybersecurityventures.com/hackerpocalypse-original-cybercrime-report-2016` (2019/01/15), 2016.

[19] Synopsys. The Heartbleed Bug. Available at `http://heartbleed.com/` (2019/06/17), 2014.

[20] CrowdStrike. VENOM. Available at `https://venom.crowdstrike.com/` (2019/06/18), 2015.

[21] Nick Wilfahrt. Dirtycow vulnerability Details. Available at `https://dirtycow.ninja/` (2019/06/18), 2016.

[22] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, Nov 1984.

[23] Ira Winkler and Araceli Treu Gomes. Chapter 5 - How to Hack Computers. pages 41–46. Syngress, 2017.

[24] Linus Torvalds. Re: [GIT PULL] usercopy whitelisting for v4.15-rc1. Available at `http://lkml.iu.edu/hypermail/linux/kernel/1711.2/01701.html` (2019/07/28), 2017.

[25] Ram Chillarege. Orthogonal defect classification. *Handbook of Software Reliability Engineering*, pages 359–399, 1996.

[26] Ram Chillarege, Wei-Lun Kao, and Richard G Condit. Defect type and its impact on the growth curve. In *Proceedings of the 13th international conference on Software engineering*, pages 246–255. IEEE Computer Society Press, 1991.

[27] Joao Viegas Carreira, Diamantino Costa, and Joao Gabriel Silva. Fault injection spot-checks computer system dependability. *IEEE Spectrum*, 36(8):50–55, 1999.

[28] Haissam Ziade, Rafic A Ayoubi, and Raoul Velazco. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186, 2004.

[29] Lena Feinbube, Lukas Pirl, and Andreas Polze. Software Fault Injection: A Practical Perspective. In *Dependability Engineering*. IntechOpen, 2017.

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

[31] Zary Segall, D Vrsalovic, D Siewiorek, D Ysskin, J Kownacki, J Barton, R Dancey, A Robinson, and T Lin. Fiat-fault injection based automated testing environment. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, page 394. IEEE, 1995.

[32] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson, and Martin Törngren. MODIFI: a MODel-implemented fault injection tool. In *International Conference on Computer Safety, Reliability, and Security*, pages 210–222. Springer, 2010.

[33] Ghani A Kanawati, Nasser A Kanawati, and Jacob A Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on computers*, 44(2):248–260, 1995.

[34] Timothy Tsai and RaviShankar Iyer. FTAPE-A fault injection tool to measure fault tolerance. In *10th Computing in Aerospace Conference*, page 1041, 1995.

[35] Seungjae Han, Kang G Shin, and Harold A Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213. IEEE, 1995.

[36] Scott Dawson, Farnam Jahanian, and Todd Mitton. ORCHESTRA: A probing and fault injection environment for testing protocol implementations. In *Computer Performance and Dependability Symposium, 1996., Proceedings of IEEE International*, page 56. IEEE, 1996.

[37] Paul D Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 379–388. IEEE, 2009.

[38] J Durães and H Madeira. Emulation of software faults by educated mutations at machine-code level. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.*, pages 329–340, 2002.

[39] João Carreira, Henrique Madeira, and João Silva. Xception: Software Fault Injection and Monitoring in Processor Functional Units. In *Proceedings of the 5th IFIP Working Conference on Dependable Computing for Critical Applications*, mar 2001.

[40] Gonçalo Pereira, Raul Barbosa, and Henrique Madeira. Practical emulation of software defects in source code. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 130–140. IEEE, 2016.

[41] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[42] Jiantao Pan. Software Testing. Available at `http://users.ece.cmu.edu/{~}koopman/des{_}s99/sw{_}testing/` (2018/12/15), 1999.

[43] Mohd Ehmer Khan and Farmeena Khan. A comparative study of white box, black box and grey box testing techniques. *Int. J. Adv. Comput. Sci. Appl*, 3(6), 2012.

[44] Mohd Ehmer Khan. *Different Approaches To Black box Testing Technique For Finding Errors*, volume 2. oct 2011.

[45] Inc. Cunningham & Cunningham. Random Testing. Available at `http://wiki.c2.com/?RandomTesting` (2018/12/19), 2014.

[46] Guru99. Boundary Value Analysis & Equivalence Partitioning with Examples. Available at `https://www.guru99.com/equivalence-partitioning-boundary-value-analysis.html` (2019/06/08), 2019.

[47] Mohd Ehmer Khan. Different approaches to white box testing technique for finding errors. *International Journal of Software Engineering and Its Applications*, 5(3):1–14, 2011.

[48] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*, volume 500. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996.

[49] Nai-Wei Lin. Data Flow Testing. Available at `https://www.cs.ccu.edu.tw/{~}naiwei/cs5812/st5.pdf` (2019/06/18).

[50] Javatpoint. Data Flow Testing. Available at `https://www.javatpoint.com/data-flow-testing-in-white-box-testing` (2019/06/24).

[51] Guru99. What is Loop Testing? Methodology, Example. Available at `https://www.guru99.com/loop-testing.html` (2019/06/08).

[52] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[53] Ron Patton. *Software testing.* Pearson Education India, 2006.

[54] Genichi Taguchi, Seiso Konishi, and S Konishi. *Taguchi Methods: Orthogonal Arrays and Linear Graphs. Tools for Quality Engineering.* American Supplier Institute Dearborn, MI, 1987.

[55] Software Testing Class. Gray Box Testing. Available at `https://www.softwaretestingclass.com/gray-box-testing/` (2018/12/21), 2012.

[56] Wenliang Du and Aditya P Mathur. Vulnerability testing of software system using fault injection. *Purdue University, West Lafayette, Indiana, Technique Report COAST TR*, pages 2–98, 1998.

[57] T Loise, X Devroey, G Perrouin, M Papadakis, and P Heymans. Towards Security-Aware Mutation Testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 97–102, 2017.

[58] M Jimenez, M Papadakis, and Y L Traon. An Empirical Analysis of Vulnerabilities in OpenSSL and the Linux Kernel. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 105–112, 2016.

[59] Tania Basso, R Moraes, Bruno P Sanches, and Mario Jino. An investigation of java faults operators derived from a field data study on java software faults. In *Workshop de Testes e Tolerância a Falhas*, pages 1–13, 2009.

[60] Plınio R S Vilela, Waldo Luis De Lucca, and Ariane Corso. Comparison of Size and Complexity Metrics as Predictors of the Number of Software Faults.

[61] M Schroeder. A practical guide to object-oriented metrics. *IT Professional*, 1(6):30–36, 1999.

[62] Rick Kuhn and Raghu Kacker. Automated Combinatorial Testing for Software. Available at `https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software` (2019/07/01), 2016.

[63] Isaac Gouy. The Computer Language Benchmarks Game. Available at `https://benchmarksgame-team.pages.debian.net/benchmarksgame/` (2018/10/31).

[64] Andreas Meyer. Random Testing. *Principles of Functional Verification*, (1):83–113, 2007.

[65] Yang Zhang Lab. What is FASTA format? Available at `https://zhanglab.ccmb.med.umich.edu/FASTA/` (2019/06/08).