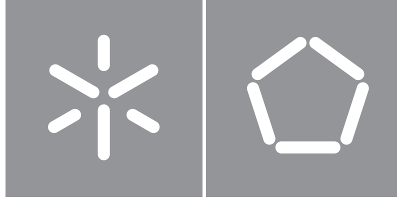
Universidade do Minho
Escola de Engenharia

Jorge Fernando Alves Cruz

Vulnerabilities Preservation Using Code Mutation

November, 2022



Universidade do Minho
Escola de Engenharia

Jorge Fernando Alves Cruz

Vulnerabilities Preservation Using Code Mutation

Master's Dissertation
Integrated Master's in Informatics Engineering

Work supervised by
Jorge Sousa Pinto
Daniela da Cruz

November, 2022

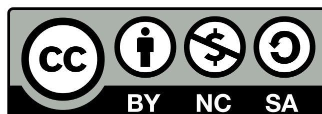
COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositóriUM of Universidade do Minho.

License granted to the users of this work



Creative Commons Atribuição-NãoComercial-CompartilhaIgual 4.0 Internacional
CC BY-NC-SA 4.0

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.pt

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

__________, ________________________________

(Location) (Date)

____

(Jorge Fernando Alves Cruz)

Acknowledgements

I would like to express my deepest gratitude to my supervisor Jorge Sousa Pinto and co-supervisor Daniela da Cruz for sticking with me during the course of this project. I could not have undertaken this journey without their invaluable patience and feedback. A big thank you to both.

Likewise, I would also like to thank my family and friends for supporting me and for insisting on asking me when this dissertation would be finished. Special thanks to my brother João for all his help and review of the document.

Abstract

The main goal of software security testing is to assess the security risks of an application so that programmers can eliminate all vulnerabilities, as early as possible, before they are exploited by attackers. There are several tools on the market that allow to perform these tests during the software development life cycle to ensure that there are no security flaws in the final product. However, like all tools, these can also have imperfections, one of them being unable to detect weaknesses in vulnerable software.

The project of this dissertation aims to tackle this problem, so that it is possible to find and correct flaws in security tests in order to, consequently, increase the effectiveness of the tools that intend to certify the security of applications. For this, the solution studied in this document is to apply syntactic transformations in vulnerable code samples without interfering in the presence of the vulnerabilities that should later be detected. This process is based on: i) code refactoring techniques that allow improving the internal quality of the software; ii) the mutation testing system used to evaluate the quality of software testing.

To implement this idea, a tool called VSG was developed with the functionality of producing new code samples with security flaws. This document describes the whole development process, from the architecture to the implementation of the tool. In the end, there is an analysis with the results obtained when trying to detect the vulnerabilities present in the samples produced through the CxSAST application of the company Checkmarx, from which this dissertation emerged.

Keywords: application security testing, code mutation, code refactoring

Resumo

O objetivo principal de testes de segurança de software consiste em avaliar os riscos de segurança de uma aplicação para que os programadores possam eliminar todas as vulnerabilidades o mais cedo possível, antes que sejam exploradas por atacantes. Existem várias ferramentas no mercado que permitem realizar estes testes durante o processo de desenvolvimento de software para garantir que não existam falhas de segurança no produto final. Porém, tal como todas as ferramentas, estas também podem apresentar imperfeições, sendo uma delas não conseguir detetar fraquezas em software vulnerável.

O projeto desta dissertação pretende combater este problema, de modo a que seja possível encontrar e corrigir falhas nos testes de segurança para, consequentemente, aumentar a eficácia das ferramentas que pretendem certificar a segurança das aplicações. Para isto, a solução estudada neste documento passa por aplicar transformações sintáticas em amostras de código vulneráveis sem interferir na presença das vulnerabilidades que deverão, posteriormente, ser detetadas. Este processo baseia-se: i) nas técnicas de refatoração de código que permitem melhorar a qualidade interna do software; ii) no sistema de testes de mutação usado para avaliar a qualidade de testes de software.

Para implementar esta ideia, uma ferramenta chamada VSG foi desenvolvida com a funcionalidade de produzir novas amostras de código com falhas de segurança. Neste documento é descrito todo o processo de desenvolvimento, desde a arquitetura até à implementação da ferramenta. No final, existe uma análise com os resultados obtidos ao tentar detetar as vulnerabilidades presentes nas amostras produzidas através da aplicação CxSAST da empresa Checkmarx, da qual esta dissertação surgiu.

Palavras-chave: teste de segurança de aplicações, mutação de código, refatoração de código

Contents

List of Figures viii

List of Tables ix

Listings x

1 Introduction 1

1.1 The Goal . . . 2

1.2 A Proposed Solution . . . 2

1.3 Document Structure . . . 2

2 Background 3

2.1 Checkmarx SAST . . . 4

3 State of The Art: Refactoring 6

3.1 Definition of Refactoring . . . 6

3.2 Refactoring Example . . . 7

3.3 Refactoring Techniques . . . 8

3.4 Automated Refactoring . . . 11

4 State of The Art: Mutation Testing 14

4.1 The Process of Mutation Analysis . . . 15

4.2 Mutation Operators . . . 17

4.2.1 Operators for Specific Programming Languages . . . 17

4.2.2 Operators for Specific Categories of Programming Languages . . . 18

4.2.3 Operators for Specific Categories of Applications . . . 18

4.2.4 Operators for Specific Categories of Bugs . . . 19

5 VSG - Vulnerable Samples Generator 21

5.1 A Samples Generation Tool . . . 21

5.2 VSG Architecture 23
5.3 Parsing with ANTLR 24
5.4 Symbol Table 26
5.4.1 Symbol Table in VSG 27
5.5 Vulnerability Flow 30
5.6 Code Transformation 32
5.6.1 Implementing a Mutation Rule 33
5.6.2 Applying the Mutations and Rules Composition 37

6 VSG Tests **40**
6.1 Original code sample vs Generated samples 40
6.2 Testing VSG against Checkmarx SAST 42

7 Conclusion **45**
7.1 Future Work 46

Bibliography **47**

List of Figures

2.1 Screenshot of CxSAST scan results, showing an SQL Injection vulnerability detected in a C# project. The Queries pane (bottom left) shows that 27 instances of the SQL Injection vulnerability were found. . . . 5

3.1 A `Stream` class hierarchy in `java.io` before and after applying **Tease Apart Inheritance** refactoring. The gray boxes represent the changes needed in each version to add the same functionality. . . . 8

4.1 Mutation Testing in Software Development . . . 14

4.2 Modern process of mutation analysis [39]. Bold boxes represent steps where human intervention is mandatory . . . 16

5.1 The inputs and the outputs of the VSG Tool . . . 22

5.2 The internal structure of the VSG Tool . . . 23

5.3 The process of a language recognizer . . . 24

5.4 The sequence of visit method calls in a parse tree . . . 25

5.5 Example of the symbol table constructed from the listing 5.3 . . . 28

5.6 Example of how the vulnerability flow will be updated after applying a transformation . . . 32

5.7 Rules Composition Schema. In this example, a list of 3 rules is applied to an original sample generating 11 different versions at the end . . . 39

List of Tables

3.1 List of primitive refactorings identified by Bill Opdyke 9

4.1 The Coupling Effect with a weaker test case set [34] 15

4.2 An Example of Mutation Operation 17

4.3 20 mutation operators proposed by Kim et al. [16] for the Java programming language . . 18

4.4 15 security-aware mutation operators proposed by Loise et al. [21] for Java 20

5.1 Attributes for each entry in the Symbol Table 27

5.2 Operations provided by the Symbol Table 29

6.1 Mutation rules used for the test 43

6.2 Results after using Checkmarx SAST to detect in the generated samples the same vulnerability present in the original 44

Listings

3.1 Code snippet before the Extract Method transformation 7
3.2 Code snippet after the Extract Method transformation 7
5.1 Grammar for traditional arithmetic expressions written in ANTLRv4 25
5.2 Example of a variable declaration 27
5.3 Example of a code sample to build the symbol table 28
5.4 Symbol Table construction when visiting a method declaration 29
5.5 Structure of the XML file with the vulnerability flow 31
5.6 Creating the Visitor for the SeparateVariableDeclarationFromAssign rule 34
5.7 The final result of the Visitor for the SeparateVariableDeclarationFromAssign rule . . . 35
5.8 The definition of the *FindAllMutations* method for the Rule class. The *Visitor* variable is the visitor created for the rule. 36
5.9 The definition of the Apply method for the Rule class 37
5.10 Creating the SeparateVariableDeclarationFromAssign rule 37
6.1 Original code sample 41
6.2 Generated code sample 42
6.3 Main program of VSG using the list of transformations in table 6.1 43

Chapter 1

Introduction

Most of today's applications are available across multiple networks and platforms and handle a lot of sensitive information, whether business or consumer. An attack on these applications can be overwhelming to the privacy of this information. A 'hacker' who manages to get into the system can not only steal passwords, financial details and other personal information, but also block users from accessing an application. These situations happen when an application has vulnerabilities. A vulnerability is a flaw in application code that can be exploited by a malicious actor and lead to a security breach.

One of the measures to avoid security flaws in a software system is to find and eliminate the vulnerabilities present in a given application. While the second part can be, in most cases, easily applied, the same cannot be said of the first part. In this context, application security testing aims to tackle the challenge of identifying security vulnerabilities. One of the techniques to perform this search is using static code analysis, which evaluates the source code of a system, without the need to execute it. On the other hand, there is dynamic analysis that takes place while the application is running and has no knowledge of how the system works internally.

Using these procedures and other techniques, it becomes possible to collect all the possibilities of attacks occurring within a project. However, one of the limitations of every searching tool is false negatives results, which in this case means results in which the vulnerability exists, but it is not detected by the analysis. This problem concerns, not only this dissertation, but also Checkmarx and other security testing solutions.

1.1 The Goal

Starting from a code sample in which it was previously confirmed to be vulnerable, it should be possible to apply mutations to the code and preserve a given vulnerability. This way, it will be possible to obtain several samples with a confirmed vulnerability. These samples will be essential to test the ability of static code analysis to detect vulnerabilities because, for each sample, a given vulnerability present in the initial code sample must be recognized. It is important that all samples produced are as diverse as possible in order to cover all possible cases of vulnerable code. This will help Checkmarx identify problems with their tool and increase their test coverage for cases where the syntax changes but the semantics remain the same.

1.2 A Proposed Solution

To achieve this goal, it's necessary to devise a tool in which the input is composed of a sample of code and a path of code elements representing the behaviour of the present vulnerability. As a result, it is expected to produce a set of mutants sharing the same security vulnerabilities from the introduced code. This tool shares features characteristic of an automated refactoring tool, but does not have rules as strict as to preserve the program's external behaviour. In addition, while a refactoring tool aims to improve software, this tool is only concerned with making transformations in any sense.

1.3 Document Structure

In addition to the introduction, which explains the context of the dissertation, this document has four more chapters. The second chapter provides a background for Checkmarx and their static analysis tool CxSAST. The reader gets to know about the company behind this project and its main tool in security testing. The next two chapters present two topics closely related to what this thesis aims to achieve. The first explains the refactoring process and its techniques, very similar to what the tool intends to do. The second topic is related to the Mutation Testing technique, which presents interesting ideas for the development. In the fifth chapter, we can find the steps taken in the development of the tool. All the details are presented from the initial parsing stage till the actual code transformation phase. From that, we get to test the tool in the sixth chapter by mutating vulnerable code samples and checking if the Checkmarx SAST tool can detect all vulnerabilities in the generated code samples. The conclusion constitutes the last chapter, together with improves and other ideas to be implemented in the future.

Chapter 2

Background

Founded in 2006, Checkmarx is a software security company headquartered in Israel. Their primary product is a platform with components of software security solutions that cover every stage of the software development life cycle, and can be easily integrated in every organization's software development process. Although their main component is the static application security testing tool (SAST), they also have open source analysis (SCA and SCS), dynamic application security testing (DAST), and other security tools related with APIs, infrastructure as code (IaC) and containers.

According to Checkmarx, the key benefits of using its software security platform are:

- Full visibility into security issues in code for both custom and open source components and better business decisions through the use of reports, customizable dashboards and APIs
- A unified central management layer for managing both organization application policies and the definition and management of all user profiles across the entire Checkmarx portfolio
- Optimization of vulnerability remediation efforts at scale by using machine learning algorithms, correlations, policy turning and custom weights to automate the scan results prioritization
- A full scope of implementation options to help customers securing their code immediately rather than going through long processes of adapting their infrastructure to a single implementation method.

2.1 Checkmarx SAST

Although the objective may also be useful for other types of security analysis, it's in the static code analysis domain that this dissertation is concerned. In this context, Checkmark SAST is a static analysis solution used to identify security vulnerabilities in custom code that can be fixed early in the software development life cycle. Supports more than 25 coding and scripting languages (Java, Kotlin, Python, JavaScript, Scala, Ruby, Swift, TypeScript, Pearl, iOS, Android, COBOL, VBScript, among others) and its frameworks without the need of extra configurations to scan any language.

Without building or compiling the source code of a software project, CxSAST works by building a logical graph of the elements and flows in the code. Against this structure, CxSAST offers an extensive list of pre-configured queries that are executed to detect known security vulnerabilities in the internal code graph for each programming language.

Using the CxSAST Auditor tool, the user can configure additional queries for security, best coding practices, QA, and business logic purposes. It can run scans and generate security reports at any given point in a software project's development life cycle.

Scan results can be obtained in CxSAST either by static reports or by an interactive interface that allows the user to observe the behavior of each vulnerability through the code. Figure 2.1 is an example of the scan results (for CxSAST v8.9.0 and up) showing an SQL Injection vulnerability.

When the user selects a specific instance of the vulnerability in the Results pane (bottom, center and right), CxSAST displays the instance's code details at the top of the pane and a path of code elements in the Path pane (top right). This path shows the flow of code elements leading from the user input to the SQL query. The user can select each element in the path which in turn CxSAST displays the element in the code context in the Source Code pane (top, left and center). The vulnerability needs to be eliminated somewhere along the path.

More information about Checkmarx and their SAST product can be found at their website [2].

Figure 2.1: Screenshot of CxSAST scan results, showing an SQL Injection vulnerability detected in a C# project. The Queries pane (bottom left) shows that 27 instances of the SQL Injection vulnerability were found.

Chapter 3

State of The Art: Refactoring

3.1 Definition of Refactoring

In his book [11], Martin Fowler defines refactoring as a process of modifying a software system in such a way that the source code is improved but the external behaviour is not changed. This term was originally introduced in 1990 by William Opdyke and Ralph Johnson [38], but it was through Martin Fowler's book that it was most represented. As a system's software evolves according to new requirements, the code becomes more and more complex and, with this, more susceptible to the introduction of bugs by programmers. According to Fowler, refactoring is the opposite of this practice. During the development of a program, its design remains firm through transformations that increase the internal quality of the software and keep the same external behaviour.

Refactoring works essentially by looking for bad smells in the code. According to Sandeep Kaur [15], bad smells represent potential problems in the code that make it difficult to understand and modify. Duplicate code is an example of a bad smell that can be solved by replacing all occurrences of the code for a function with the same functionality. Not only is the code easier to understand but also more sustainable and with better quality. Furthermore, refactored code reduces the code maintenance cost and the chance of introducing bugs [14].

Other definitions of refactoring are more abstract. In his dissertation [41], Donald Roberts defines refactoring as a pair (pre, T), where pre is the set of pre-conditions that a program must satisfy to perform a transformation, and T corresponds to the transformation. This definition emerged so that transformations that do not preserve external behaviour were also included.

3.2 Refactoring Example

The impact of a transformation on the program differs from the type of refactoring applied. While some refactorings make localized changes to the code, others operate at higher levels and in a larger scale [31]. A common example of a localized transformation is the extraction of a code snippet to its own method.

Listing 3.1: Code snippet before the Extract Method transformation

```
public void printNumberOfLines(String filePath) {
   int noOfLines = 0;
   LineIterator lineIterator = FileUtils.lineIterator(new File(filePath));
   while (lineIterator.hasNext()) {
      lineIterator.nextLine();
      noOfLines++;
   }
   System.out.println("Number of lines: " + noOfLines);
}
```

When applying the **Extract Method** refactoring to the code fragment corresponding to the functionality of calculating the number of lines in a file, the final result is:

Listing 3.2: Code snippet after the Extract Method transformation

```
public void printNumberOfLines(String filePath) {
   int noOfLines = numberOfLines(filePath);
   System.out.println("Number of lines: " + noOfLines);
}

public int numberOfLines(String filePath) {
   int noOfLines = 0;
   LineIterator lineIterator = FileUtils.lineIterator(new File(filePath));
   while (lineIterator.hasNext()) {
      lineIterator.nextLine();
      noOfLines++;
   }
   return noOfLines;
}
```

With a simple change, not only we made the methods more refined, but also increased the chances of being reused by others. The opposite refactoring (**Inline Method**) also exists and can be useful when we have a method whose body is as clear as the method name. Therefore, it is necessary to evaluate the context to know if a refactoring will have a positive or negative impact on the code.

In terms of global transformations, we can look at the effect of Fowler's **Tease Apart Inheritance**

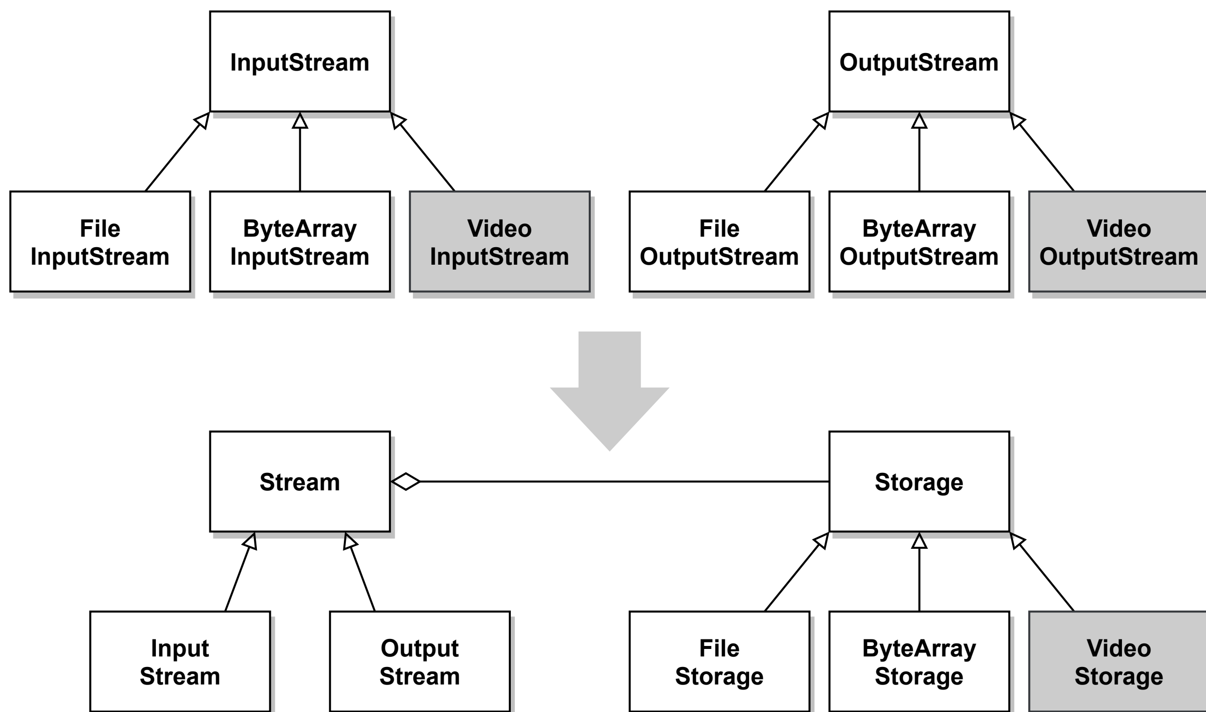


Figure 3.1: A `Stream` class hierarchy in `java.io` before and after applying **Tease Apart Inheritance** refactoring. The gray boxes represent the changes needed in each version to add the same functionality.

refactor. The top of figure 3.1 shows a hierarchy of classes from `java.io` for streaming data. Suppose we want to add the features of reading and writing to a video stream with this structure. Unfortunately, it would be difficult because we would have to add two new classes, `VideoInputStream` and `VideoOutputStream`, probably with duplicate code between the two. The problem with this class hierarchy is the mixture of two concerns: the direction of the flow (input or output) and the type of storage in which it is performed. Not only is it difficult to make changes, as the resulting code is hard to understand.

However, we can solve this inconvenience by applying the refactoring **Tease Apart Inheritance**, in order to create two different hierarchies, as we can see at the bottom of figure 3.1. With this structure, to add streaming video, all we need to do is add the `VideoStorage` class as a subclass of `Storage`.

3.3 Refactoring Techniques

In 1992, William Opdyke developed the first detailed refactoring elaboration in his doctoral thesis. In this dissertation [37], he presented 23 primitive refactorings, shown in the table 3.1, together with a set of preconditions for each one that would ensure the behaviour of the software after the transformation. This collection was achieved by looking at various systems and pointing out the types of refactorings that would apply in object-oriented programming. All this work made it possible for other elaborated techniques to emerge in the refactoring field.

Table 3.1: List of primitive refactorings identified by Bill Opdyke

creating an empty class
creating a member variable
creating a member function
deleting an unreferenced class
deleting an unreferenced variable
deleting a set of member functions
changing a class name
changing a variable name
changing a member function name
changing the type of a set of variables and functions
changing access control mode
adding a function argument
deleting a function argument
reordering function arguments
adding a function body
deleting a function body
convert an instance variable to a variable that points to an instance
convert variable references to function calls
replacing statement list with function call
inlining a function call
change the superclass of a class
moving a member variable to a superclass
moving a member variable to a subclass

A refactoring technique refers to a procedure in which a set of steps is taken in order to achieve the desired transformation. After identifying where the software has bad smells, it is necessary to determine which refactoring techniques to apply to keep the source code as clean as possible. In Martin Fowler's book [11] we can find a long catalog of refactorings techniques and, for each one, their steps and motivation. This list is divided into 8 groups.

Composing Methods

The composition of methods is one of the most common groups in refactoring techniques. The objective is, in most cases, to restructure long and difficult to understand methods. For example, the refactoring **Extract Method** shown above is one of the techniques belonging to this group. Also, when we want to compose methods, one of the biggest concerns corresponds to temporary variables, so this group also includes techniques related to these variables and the parameters of the methods.

Moving Features between Objects

These refactorings are used to decide where to place responsibilities between different classes in order to make the best decisions in object design. Through them we can move functionality between classes, create new classes and hide the implementation details from public access. Most problems can be solved with the **Move Method** and **Move Field** refactorings, as well as the **Extract Class** that allows you to create a new class from another.

Organizing Data

In object-oriented programming, one of the most important aspects is the definition of new types from primitive types. This characteristic, despite being fundamental in problem-solving, brings, in itself, an increase in potential disorders to keep the data structure clean as complexity increases in level. These refactoring techniques make handling information easier. For example, when we have a simple information value that would be more useful as an object, we can apply the **Replace Data Value with Object** refactoring to create a new class that stores the value and all its behaviour.

Simplifying Conditional Expressions

As the name implies, these refactoring techniques are intended to facilitate the logic of conditions that, at times, becomes complicated and confusing. The main refactoring is **Decompose Conditional** which allows you to extract all parts of a complicated conditional sentence (if-then-else) in separate methods. This makes the code more readable and easier to maintain by a different programmer.

Simplifying Method Calls

These techniques are related to the importance of interfaces in the development of object-oriented programs. One of the simplest techniques is to rename methods, variables and classes so that the code is as readable as possible. Other concerns include the method parameters, the class constructors and the error handling mechanism.

Dealing with Generalization

Generalization raises a set of refactoring techniques that influence functionalities along a class hierarchy. The most common techniques in this group are to promote and descend methods or instance variables among classes in the hierarchy. In addition to moving functions, there are also techniques, such as **Extract Subclass**, **Extract Superclass** and **Extract Interface**, that change the hierarchy by adding

new elements at various points. In case there are unnecessary classes in the hierarchy, we can use the refactoring **Collapse Hierarchy** to remove them.

Big Refactorings

In his last group, Fowler presents 4 major refactorings. While the latter groups consisted of individual refactoring changes, this group intends to ”play”at a higher level through a lot of transformations that take longer to finish. One of the refactoring techniques included corresponds to the one presented in section 3.2, which, as we have seen, exerts an enormous change in terms of the program structure.

3.4 Automated Refactoring

Although it is possible to apply refactorings manually, tool support is considered crucial. During refactoring, users need to synthesize and analyze large collections of code to identify bad smells, decide the best solutions to eliminate undesirable characteristics and apply potentially complex and error-prone transformations without changing the external behaviour of the system. If we take, as an example, the simple renaming of a method, the user needs to propagate this change throughout an entire system. This becomes tedious and costly when developing a project, making automated refactoring software essential in a time-consuming and error-prone manual task. Nowadays, a wide range of tools is available that automate various aspects of refactoring [18].

The refactoring process is divided in distinct activities that can be supported by different tools:

1. Analyze the software to determine where it should be refactored (detecting bad smells).
2. Determine which refactoring(s) should be applied to the identified places in order to remove the code smells.
3. Assure that the applied refactoring preserves the external behaviour of the software.
4. Apply the refactoring.
5. Evaluate the result of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort).

“Fully-automatic tools complete the multi-stage refactoring process without user interaction, from the initial identification of where it is needed through the selection and application of a specific refactoring.” [24] Although these tools assist software developers, the absence of user input along the steps is the root of worthless identifier names, lack of customizability, and negative impacts on a user’s understanding of a system [4, 9]. On the contrary, semi-automated refactoring tools attempt to address these problems

by using user input to value the refactoring process whilst automating the tedious, error-prone and complex sub-tasks. This single-stage focus makes the balance between automated and manual tasks essential to readable and well-designed code. Examples of semi-automated refactoring tools include Eclipse's refactoring transformation support, and jDeodorant's code smell detection.

In the next sections, I will explore the different characteristics that affect the usability of an automated refactoring tool and that can be interesting for the tool we want to create in this dissertation.

Reliability

The reliability of a refactoring tool is related to the ability to ensure that, after a transformation, the external behaviour of the program remains similar. Most tools provide this feature by checking a set of preconditions before applying the refactoring and running tests on the system's functionalities after applying the refactoring. In this dissertation, the reliability of the tool will be more relaxed since the objective is to preserve a vulnerability of the system and not its external behaviour.

Configurability and Openness

Most refactoring tools are integrated into industrial IDEs through their extensibility mechanisms that sometimes make it impossible to configure the tools with specific user preferences. Some of the attributes that a user should be able to control are:

- refactorings and bad smells specifications,
- the link between bad smells and refactorings [45],
- definitions of composite refactorings from primitive ones [33].

For this dissertation, having an open configurable tool is essential to control the different versions we can produce from a code sample.

Coverage

This attribute is fundamental for refactoring tools because it indicates how comprehensive the tool is in terms of the types of activities it can perform. The more complete the tool is, the easier it is for the user to improve the code. Unfortunately, most tools tend to focus on just one aspect of the refactoring process.

Scalability and Performance

The ability to combine a sequence of primitive refactorings into a composite refactoring increases the scalability and performance of a refactoring tool. This characteristic allows solving problems at higher

levels and, in the case of this dissertation, it allows transformations on larger scales. The performance gain is related to the fact that the tool only needs to check the preconditions of the composite refactoring once, instead of checking separately for each primitive refactoring in the sequence [25, 41].

Language Independence

Like coverage, the tool's range for different programming languages should be as complete as possible and in addition, the tool should provide the necessary hooks to add language-specific behaviour [27]. There are several techniques for achieving language independence:

- *Meta modelling* is a technique presented by both Tichelaar et al. [44] and Mens et al. [26]
- The notion of *generic program refactoring* introduced by Lämmel [22]
- Ward and Bennett suggested translating the code into the formal intermediate language WSL, where it can be restructured, refined and abstracted, and then translated back into the original language [48]. For any language, just create an automatic translator in both directions for WSL.

Chapter 4

State of The Art: Mutation Testing

Mutation Testing is a fault-based testing technique whose purpose is to measure the adequacy of an input software test set in terms of its ability to detect certain types of faults. It was first developed and published in the late 1970s by DeMillo, Lipton and Sayward [5]. By applying simple syntactic transformations to the original program, this technique pretends to simulate errors that programmers often make during software development. Each generated version is called a mutant and is based on a well-defined mutation operator such as the replacement of an arithmetic operator in the original program with other operators, which represents a programmer using a wrong arithmetic operator. When evaluated by a test set, if his behavior is different from the original program, the mutant is said to be killed. Otherwise, it remains alive because either it is equivalent to the original program (same functionality although syntactically different), or the test set is inadequate to detect the fault and must be improved by adding test cases to kill the live mutant.

Figure 4.1: Mutation Testing in Software Development

Although there are a huge number of potential failures in a given program, the mutation testing technique selects only a subset of mutation operators to generate mutants, close to the original version, hoping that these will be enough to represent all faults. This theory is based on two hypotheses, both introduced and proposed by DeMillo et al. in 1978 [5]: the Competent Programmer Hypotheses and the Coupling Effect. The first hypothesis states that programmers are competent and tend to write programs that are close to being correct, which means most software faults introduced are due to small syntactic errors that can be easily corrected with small changes. The second hypothesis asserts that if a test set can be so

sensitive as to distinguish the original program from those generated by only simple errors, then it will also distinguish from those created by more complex errors.

Later, Offutt [35] extended this hypothesis into the Coupling Effect Hypothesis and the Mutation Coupling Effect Hypothesis with a definition of simple and complex faults. A simple fault consists on creating a single mutant by making a single syntactical change, while a complex fault is represented as a higher-order mutant which is generated by introducing multiple mutations into the program. With this, his definition of the Coupling Effect states that ”complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults” [35]. For the empirical analysis, this was restricted into the Mutation Coupling Effect Hypothesis: ”Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will also detect a large percentage of the complex mutants” [35].

Many research works [20, 30, 34, 35] were done in order to validate the coupling effect hypothesis. Offutt [34, 35] performed an experiment where it was generated all possible second-order mutants (mutants created by 2 mutations) of three programs, MID, TRITYP, and FIND. The results in table 4.1 shows that a weaker test case set, generated to target a random selection of 1-order mutants, is very successful at killing a higher percentage of 2-order mutants, which implies that the mutation coupling effect hypothesis does indeed manifest itself in practice.

Table 4.1: The Coupling Effect with a weaker test case set [34]

Program	Mutation order	M	K	Eq	Live	MS
MID	1-order	196	161	16	19	.89
	2-order	19110	18607	145	503	.98
MID	1-order	196	161	16	44	.76
	2-order	19110	17276	145	1689	.91
TRYTIP	1-order	970	745	110	115	.87
	2-order	469965	443875	2154	23936	.95
TRYTIP	1-order	970	636	110	224	.74
	2-order	469965	426030	2154	46781	.90

4.1 The Process of Mutation Analysis

This section presents a detailed view of the modern mutation testing process. This process forms an extension of the one proposed by Offutt and Untch [36] and is based on the latest advances in the area. Performing the mutation analysis for a given P program, starts by selecting a set of transformation rules to apply in the original program and then generating the actual executable programs. The next step concerns about some problematic mutants that need to be removed. For example, there are some mutants that

cannot be killed by testing because, although syntactically different, they produce the same output as the original program (equivalent mutants). There are also some redundant mutants, i.e., mutants that are killed when other mutants are killed, which must be removed due to the fake impact they have on the mutation score and which can lead to tests of a lower quality than intended.

With the mutants all generated, the next step is to create the test cases with the potential to kill all the mutants, run them for each alternative program and determine how well they scored. This score, known as the Mutation Score, indicates the quality of the input test set and is calculated with the ratio of dead mutants over the total number of live mutants. At this point, a reduction may be applied to the test suite by removing tests that are potentially ineffective. Also, at the same time, the tests can be arranged so that those with the most impact are executed first. These steps are repeated until the tester is satisfied with the mutation score. A mutation score threshold can be set as a policy decision to require testers to test software to a predefined level.

The last step of this process evaluates the original program by comparing the results of the test executions with those expected. If failures are detected then the programmer must find the faults in the program, repair these problems and relaunch the process until reaching an acceptable mutation score and a faultless program.

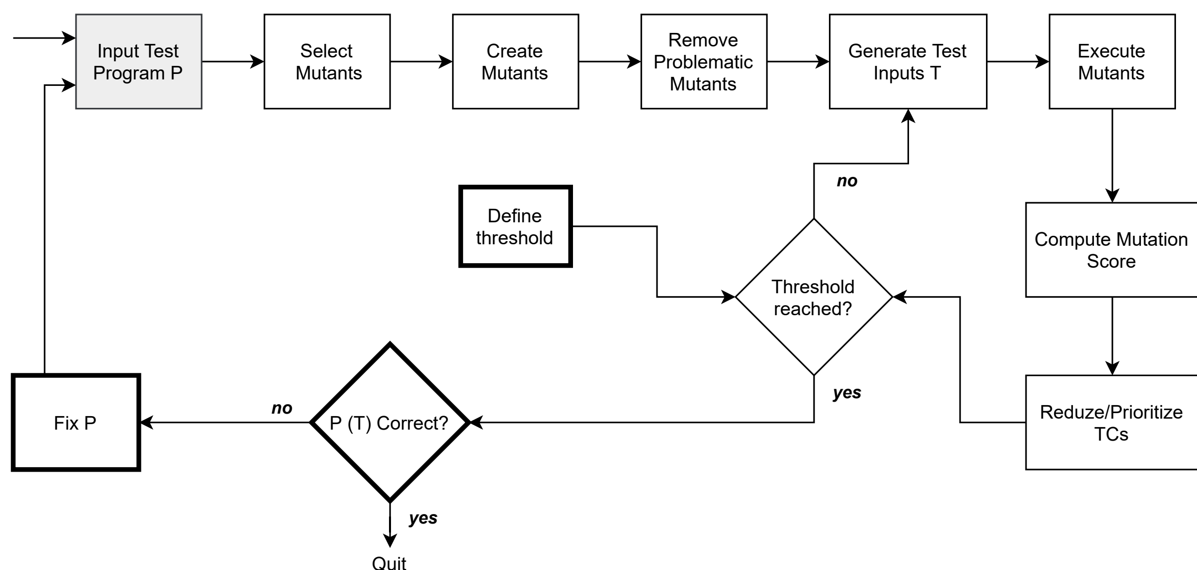


Figure 4.2: Modern process of mutation analysis [39]. Bold boxes represent steps where human intervention is mandatory

4.2 Mutation Operators

A transformation rule that generates a mutant from the original program is known as a mutation operator. Table 4.2 shows the mutant p', generated by changing the *and* operator (&&) of the original program p, into the *or* operator (||), thereby producing the mutant p'. A large amount of work has focused on designing mutant operators that target different (categories of) programming languages, applications, types of defects, programming elements, and others.

Table 4.2: An Example of Mutation Operation

Program p	Mutant p'
... if (a > 0 && b > 0) return 1; ...	... if (a > 0 || b > 0) return 1; ...

4.2.1 Operators for Specific Programming Languages

King and Offutt [17] introduced the first set of formalized mutation operators for the Fortran programming language. These typical mutation operators were implemented in the Mothra mutation system.

In 1989, Agrawal et al. [1] proposed a comprehensive set of mutation operators for the ANSI C programming language. There were 77 mutation operators defined in this set, which was designed to follow the C language specification. These operators are classified into variable mutation, operator mutation, constant mutation, and statement mutation.

Kim et al. [16] were the first to design mutation operators for the Java programming language. They proposed 20 mutation operators (Table 4.3) for Java using a safety technique which investigates and records the result of system deviations. Based on the results, 20 Java mutation operators were designed, falling into six groups: Types/Variables, Names, Classes/interface declarations, Blocks, Expressions, and others.

Derezinska and Kowalski [8] introduced six object-oriented mutant operators designed for the intermediate code that is derived from compiled C# programs. Their work also revealed that mutants on the intermediate language level are more efficient than the high-level source code level mutants.

Mirshokraie et al. [28, 29] proposed a set of JavaScript operators. These are designed to capture common mistakes in JavaScript (such as changing the *setTimeout* function, removing the *this* keyword, and replacing *undefined* with *null*). Experimental results indicate the efficiency of these operators in generating non-equivalent mutants.

Table 4.3: 20 mutation operators proposed by Kim et al. [16] for the Java programming language

Category	Operators
Types and variables	Type replacement operator Variable replacement operator
Names	Modifier change operator
Classes/interfaces declarations	Package replacement operator Imported type replacement operator Cluster structure change operator Constructor declaration change operator Formal parameter change operator
Blocks	Control-flow disruption operator Scope change operator Statements swap operator Exception handler change operator
Expressions	Language operator replacement Literal change operator Constructor replacement operator Accessed Accessed field replacement operator Method name replacement operator Argument Argument change operator
Others	Coverage operator Equivalent replacement operator

4.2.2 Operators for Specific Categories of Programming Languages

Derezinska and Kowalski [8] explored and designed the mutant operators for object-oriented programs through C# programs. They argued that traditional mutant operators are not enough to reveal object-oriented flaws. Hu et al. [13] studied in depth the equivalent mutants generated by object-oriented class-level mutants and revealed differences between class-level and instruction-level mutation: instruction-level mutants are easier to be killed by test cases.

Aspect-Oriented Programming is a programming paradigm that aids programmers in separation of cross-cutting concerns. Ferrari et al. [10] proposed 26 mutation operators based on a generalization of faults for general Aspect-Oriented programs. These mutation operators are divided into three groups: point cut expressions, aspect declarations, and advice definitions and implementation. This work uses AspectJ as a representative of aspect-oriented programs.

4.2.3 Operators for Specific Categories of Applications

Deng et al. [6, 7] defined mutant operators specific for the characteristics of Android apps, such as the event handle and the activity lifecycle mutant operators. Usaola et al. [46] introduced an abstract

specification for defining and implementing operators for context-aware, mobile applications. Similarly, Linares-Vásquez et al. [19] introduced 38 mutation operators for Android apps. These operators were systematically derived by manually analysis of types of Android faults.

In 2014, Maezawa et al. [23] proposed a mutation-based method for validating Asynchronous JavaScript and XML (Ajax) applications. The approach is based on delay-inducing mutant operators that attempt to uncover potential delay-dependent faults. The experimental study suggests that by killing these mutants, actual errors can be revealed.

4.2.4 Operators for Specific Categories of Bugs

Brown et al. [3] proposed a technique to find mutation operators from source code repositories with the intuition of getting mutants semantically similar to real faults. Loise et al. [21], concerning security issues, proposed 15 security-aware mutant operators for Java. Nanavati et al. [32, 49] realized that few operators are able to simulate memory faults, so they proposed 9 memory mutant operators targeting common memory faults. Garvin and Cohen's work [12] focus on feature interaction faults. An exploratory study was conducted on the real faults from two open-source projects and mutants are proposed to mimic interaction faults based on the study's results.

Other examples are mutation operators that target specific C program defects or vulnerabilities. Shahriar and Zulkernine [43] proposed 8 mutation operators to generate mutants that represent Format String Bugs (FSBs). Vilela et al. [47] proposed 2 mutation operators representing faults associated with static and dynamic memory allocations, which were used to detect Buffer Overflows (BOFs). This work was subsequently extended by Shahriar and Zulkernine [42], who proposed 12 comprehensive mutation operators to support the testing of all BOF vulnerabilities, targeting vulnerable library functions, program statements, and buffer size.

Table 4.4: 15 security-aware mutation operators proposed by Loise et al. [21] for Java

Use predictable pseudo random number generator
Remove path traversal sanitization
Use weak message digest
Remove host name verification
Make XML parser vulnerable to XML Entity Expansion attack
Make XML parser vulnerable to XML eXternal Entity attack
Remove encryption in socket
Unsecure cookie
Remove HTTP-only flag from cookie
Use RSA with short key
Use Blowfish with short key
Permit SQL injection
Use DES in symmetric encryption
Use ECB in symmetric encryption
Remove regex sanitization

Chapter 5

VSG - Vulnerable Samples Generator

This chapter represents the core of this dissertation by introducing the VSG tool, together with all the steps and decisions taken during its development. It is in this section that we can find a path to the solution of the problem presented at the beginning, describing the whole development process from the tool architecture to its implementation. Within the topics presented in the state of the art, this tool will focus on the application of transformations in the code, both present in refactoring 3 and mutation testing 4. The most important point will be how syntactic transformations will be applied to vulnerable code samples in order to obtain the greatest possible diversity regarding the presence of vulnerabilities.

Therefore, the first part of this chapter gives an introduction to the tool by describing what it is intended to achieve and how it will be achieved. The second section explores the tool's architecture, explaining how it is structured and what technologies have been used. Finally, the remaining sections present the main components of the tool and the implementation of syntactic transformations on code samples.

5.1 A Samples Generation Tool

As previously mentioned, the main objective of this project is the development of a tool that generates code samples with a security flaw that is known by the user. The path chosen to achieve this goal involves an initial sample of code, provided by the user as input, with a known vulnerability. The tool must then apply several syntactic transformations to the sample code without losing the security flaw, thus obtaining a set of code samples sharing the same characteristic: having the same vulnerability 5.1.

To ensure that all the transformations applied to the code have an impact on the vulnerability, the user must also provide a metadata file with information about the flow of the vulnerability in the code sample. This file will be an XML file containing a path with all the nodes that define the behavior of the security

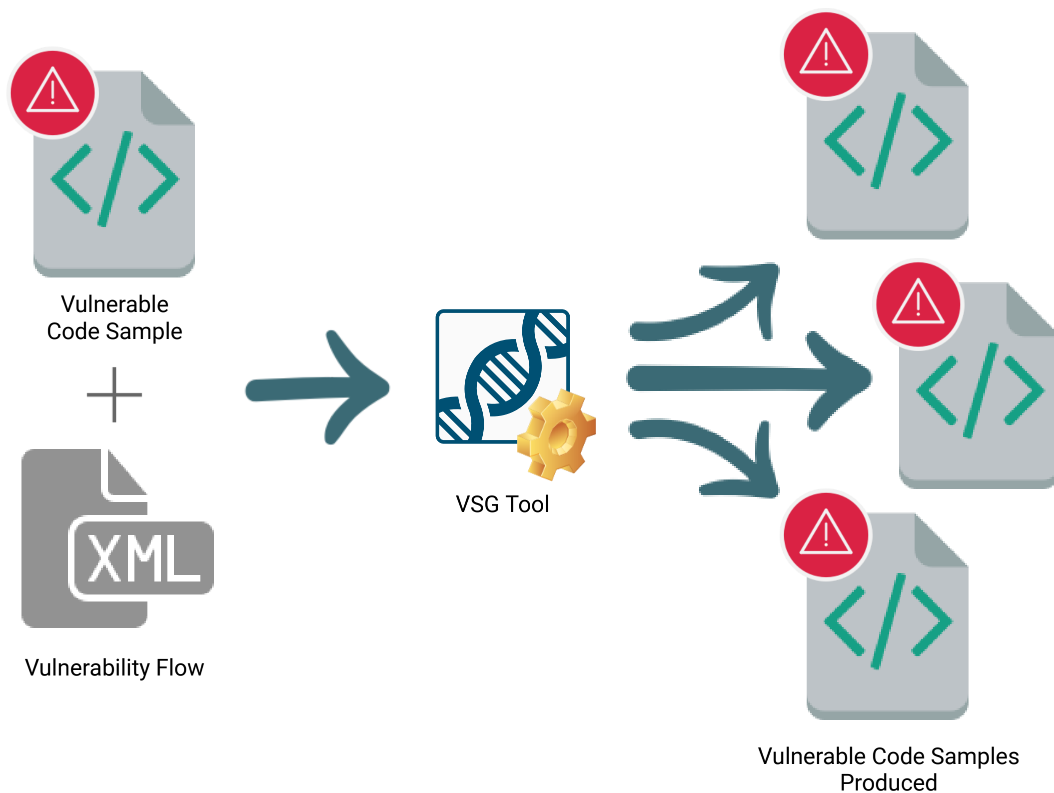


Figure 5.1: The inputs and the outputs of the VSG Tool

flaw. If the transformations were applied anywhere in the code, the result would be a large percentage of the code samples produced with an identical vulnerability, in terms of behavior, to the one present in the initial sample, so we can assume that its presence would not be difficult to detect since we know that the initial vulnerability was also detected at the beginning (remember the main problem of this project).

Each type of syntactic transformation that can be applied over the code will have the name of mutation rule. One of the key aspects for this tool will be how different mutation rules can be combined in order to vary the set of code samples obtained at the end of the process. This feature allows exploring the multiple case where a vulnerability is present, making the role of a security testing tool more difficult to assume. In addition, a mutation rule can be applied at various points in a code sample, so this aspect will also be used in favor of the tool.

The initial step of the process is to parse the sample code in order to identify the places where the transformations can be applied. For this purpose, the ANTLR parser generator was used for the lexical and syntactic analysis of the code, which will be described in the section 5.3. As for the language, the tool will be written in C#, an object-oriented language that easily links ANTLR to the development of the mutation rules and which the author is comfortable to work with. In this first version, only code samples written in the Java language will be accepted as input for the tool. This choice is due to the fact that Java is one of the most used languages by programmers and therefore needs more concern from security testing applications.

5.2 VSG Architecture

Before jumping to the implementation, this section will explain how the tool is structured internally and what is the interaction between the different components that allow the whole process to be executed. The figure 5.2 introduces an abstract overview of this structure through a diagram with the main VSG components. Although there are other elements in this structure, such as the component that allows loading the code samples together with their vulnerability flows, the most important ones that allow the magic to happen are represented in the diagram.

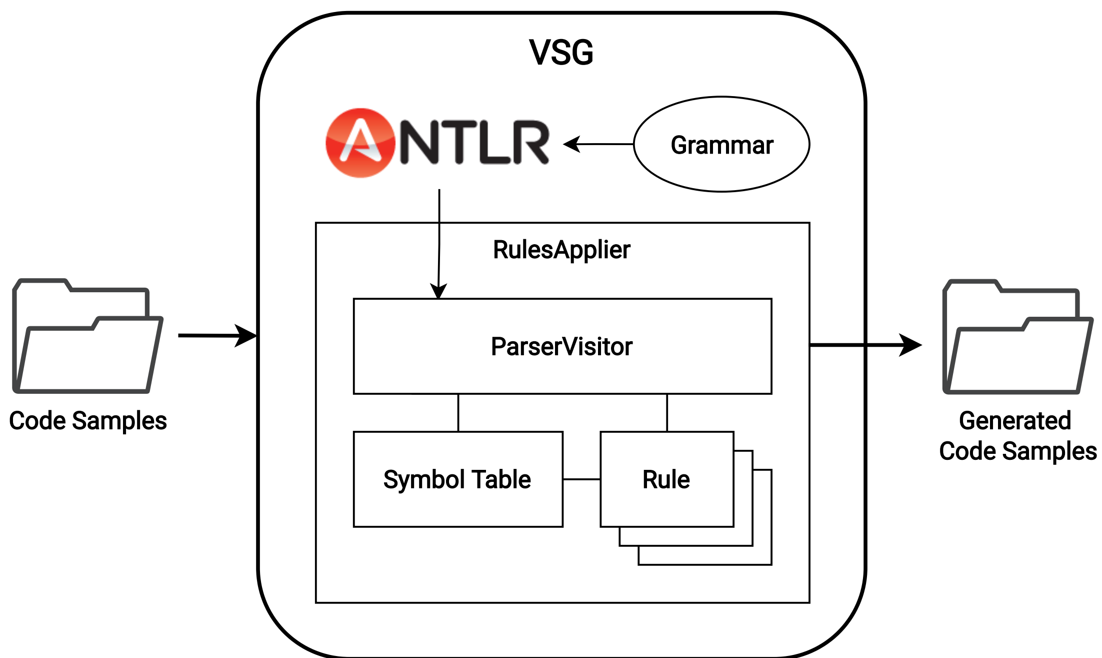


Figure 5.2: The internal structure of the VSG Tool

As stated in the previous section, it is necessary to analyze the code samples in order to identify the places where a mutation rule can be applied. In this sense, through the grammar that specifies the language in which the samples are written, the application uses ANTLR to generate a recognizer for that language. The grammar only needs to be compiled once to create a parser written in C#. This way, ANTLR replaces the work of having to create a parser manually that can be used by the mutation rules. This parser, which will be explained further in section 5.3, constitutes one of the components of this structure.

The next component represents the mutation rule itself. Each rule will be a component that will use the parser to find all the places where it can be applied and collect all the applicable transformations on the code sample. The way how all possible locations will be detected and transformations executed represents the main focus of this dissertation and will be described in detail in section 5.6. The set of components consisting of the mutation rules and the parser represents the *RulesApplier* which, fed by the initial code samples, will generate the output with the code samples produced.

Finally, the whole structure was assembled to obtain the initial code samples from a directory instead of receiving one at a time to execute the whole process. For this, the directory must follow a predefined structure so that the tool can correctly load the code samples together with the corresponding vulnerability flows. In the end, all the code samples produced will be exported to another directory that will also follow a structure that facilitates the correspondence between the initial code sample and those that were generated from it.

5.3 Parsing with ANTLR

The development of a parser for a certain language can be a difficult and time-consuming task, requiring several skills before starting to write one. For this purpose, Terence Parr, together with other colleagues, created ANTLR (ANother Tool for Language Recognition) to speed up this process. This tool receives as input a grammar specification and automatically produces a parser that can build and walk parse trees. No effort is required in the analysis of structured texts such as program source code. This tool is an important part of the VSG and this section tries to explain a little of its process and why it is important for the mutation rules.

For ANTLR to work, it is necessary to provide a grammar with two types of rules that describe the language syntax: parser and lexer rules. Lexer rules are responsible for lexical analysis, i.e. the process of identifying tokens in a character stream. Parser rules focus on recognizing the structure of the input sentence by building the abstract syntax tree from the identified tokens. Diagram 5.3 demonstrates the process of a language recognizer for an arithmetic expression with an example. By entering the input sentence *5+4*, the lexer will recognize the *INT* (integers) and *PLUS* tokens and feed them to the parser, which in turn, will produce a parse tree with the arithmetic expression structure.

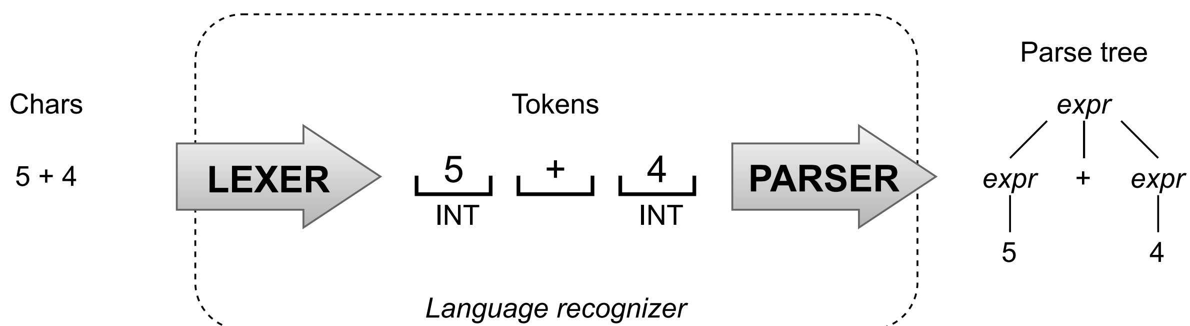


Figure 5.3: The process of a language recognizer

A grammar must follow the syntax defined by ANTLR to specify other languages. Listing 5.1 is an example of a grammar for traditional arithmetic expressions (addition, subtraction, multiplication and division) with the name *Expr*. This language has only one parser rule called *expr* with four alternatives to define an arithmetic expression. The remaining rules are lexer rules intended to recognize operators

(*MUL*, *DIV*, *ADD*, *SUB*) and integers (*INT*) with one or more digits. Note that all parser rules start with a lowercase letter and all lexer rules start with an uppercase letter.

Listing 5.1: Grammar for traditional arithmetic expressions written in ANTLRv4

```
grammar Expr;

expr
   : expr (MUL|DIV) expr
   | expr (ADD|SUB) expr
   | INT
   | '(' expr ')'
   ;

ADD : '+';
SUB : '-';
MUL : '*';
DIV : '/';
INT : [0-9]+ ;
```

The parse tree is a useful data structure that contains complete information on how the parser has grouped the recognized tokens. From another point of view, when analyzing a program source code, the result would be a tree with an easy to process structural representation of the program. However, a mechanism is still needed to walk this tree in order to take the desired actions at the correct nodes. One of the tree-walking mechanisms that ANTLR offers is tree visitors.

When provided with a grammar, ANTLR is able to generate a visitor interface with a visit method for each rule that will be called along a depth-first walk on the parse tree. In other words, this mechanism follows a visitor pattern on the parse tree. As an example, figure 5.4 demonstrates the order of method calls among the visitor methods for the parse tree obtained in the input sentence *5+4*.

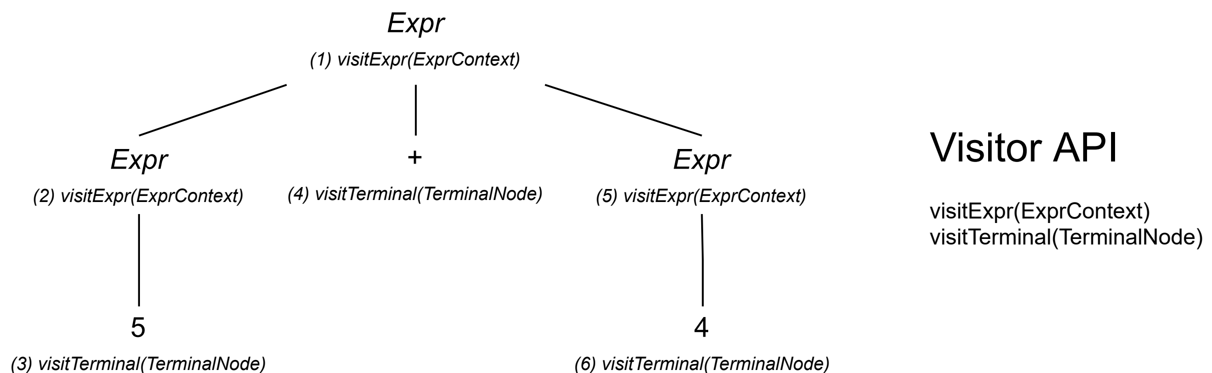


Figure 5.4: The sequence of visit method calls in a parse tree

Each visit method will take as argument a *context* object. According to Terence Parr, "each context

object knows the start and stop tokens for the recognized phrase and provides access to all elements of that phrase" [40]. This means that for the example in figure 5.4, the first *ExprContext* has access to the expression subtrees.

For the matter of this project, the *ParserVisitor* generated by ANTLR for each programming language will be an important component for the mutation rules. Each rule needs to explore the structure of a program in order to evaluate and apply the transformations in the right places. Therefore, *ParserVisitor* enables rules to take actions at the correct nodes of a program's parse tree. For example, imagine a rule that wants to make changes to if statements in a program. All it needs is to override the *visitIfStatement* method (assuming the grammar contains a parser rule for the if statement) provided by the *ParserVisitor* and explore the context to find out how the transformation will be applied.

Having the ability to parse the code, the next step was to start implementing the rules. However, it was quickly realized that a data structure would be needed to assist the rules in code analysis and correct application of transformations. Thus, the next section presents the symbol table.

5.4 Symbol Table

One of the problems that arose during the implementation of the mutation rules was the situation when new identifiers were added within a scope. If, for example, a mutation rule adds a new variable within a method, it is necessary to verify that the name of that variable is not already being used within the scope of the method. Otherwise, the result would be conflicts of identifiers, which in turn would result in the production of non-compilable code samples.

In addition to this problem, other complications emerged, such as mutation rules that needed to infer the type of an identifier so that its transformation could be successfully applied. In this context, it was important to implement an auxiliary data structure with all the information about the symbols and scopes defined in the source code of a program. Being widely used in compiler design to achieve compilation time efficiency, the symbol table proved to be the solution to these problems.

A symbol table consists of a data structure with the purpose of storing identifiers (symbols) present in the source code of a program, such as classes, methods and variables, and associating to each one its information and relationship with other identifiers. There are several types of data structures that can be used to implement a symbol table, the most commonly used being lists, hash tables and trees. The main purposes of a symbol table are:

- Inference of an identifier type
- Verification of the existence of a symbol within a scope
- Getting the scope of a symbol

The two most common operations in a symbol table are:

insert() Used during the code analysis to insert a new entry in the table with the symbol identifier and its information.

lookup() Used to search for a name in the symbol table and, if it exists, return its information.

5.4.1 Symbol Table in VSG

The first step in implementing the symbol table used in the VSG tool was choosing the information that would be added for each entry in the table. This decision was made by the mutation rules that were implemented, which means that new mutation rules in the future may require more information about each symbol. Each entry contains the information present in the table 5.1. For example, for the next variable declaration, the entry stored in the symbol table would be: `<”firstName”, Variable, String>`.

Listing 5.2: Example of a variable declaration

```
String firstName = ”John”;
```

Table 5.1: Attributes for each entry in the Symbol Table

Name	Symbol’s identifier
Kind	Represents the symbol’s kind in the program (e.g. Class, Variable, Parameter, Method)
Type	Stores the symbol’s type (e.g., int, float, boolean, String)

The next decision was how the scopes would be defined and represented in the symbol table. Taking advantage of the rich C# library, each scope became a dictionary in which the key consists of the symbol identifier and its value represents the additional information of the symbol. In addition, each scope also has access to the scope in which it is contained so that it is possible to return to the previous scope. The reason for this is that to implement the *lookup* operation it is necessary to look for the symbol in the outer scopes in case it is not present in the current one. Each symbol will thus be associated with a scope that will only be initialized if it is appropriate to its kind. For example, a symbol that represents a variable does not need a scope.

Let’s look at the following example: figure 5.5 shows the symbol table represented from the sample code present in listing 5.3. This sample consists of a simple class with two fields and a method to get the full name of a person. By looking at the table, we can confirm that certain kinds of symbols have no scope, in this case the *Field* and *Variable* kinds. We can also verify that the *Class* kind entry does not have a defined type since the symbol itself is a type. Finally, as mentioned, we can see that from each scope it is possible to go back to the outer scope, except for the root scope.

Listing 5.3: Example of a code sample to build the symbol table

```
class Person {

   private String firstName;
   private String lastName;

   public String getFullName() {
      String fullName = firstName + " " + lastName;
      return fullName;
   }
}
```

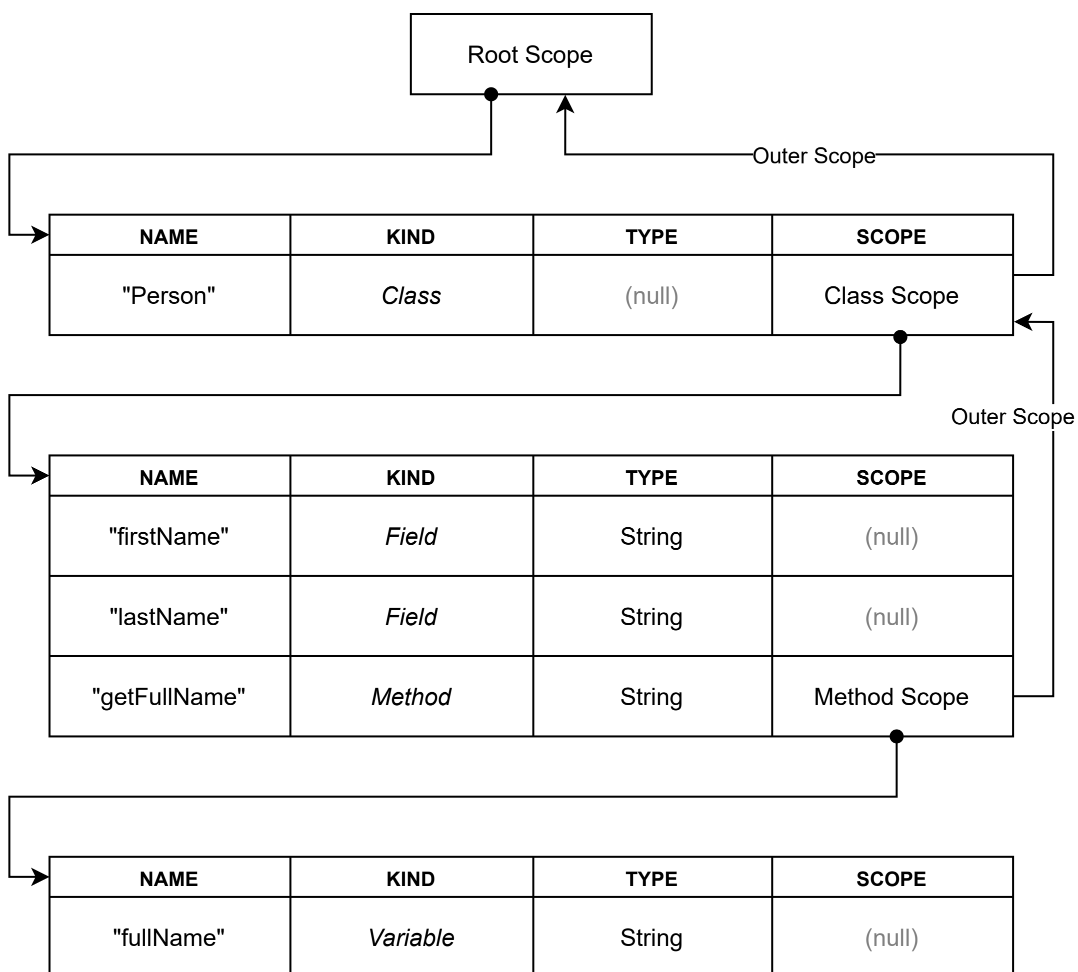


Figure 5.5: Example of the symbol table constructed from the listing 5.3

The symbol table will then be defined by the root scope, the scope it is currently in, and the last added symbol that will be used during construction to know where the next scope will be open. As for its operations (Table 5.2), the symbol table built for VSG presents operations for the construction phase (*OpenScope*, *AddSymbol* and *ExitScope*) and for the exploration phase (*EnterScope*, *ExitScope*, *Lookup* and *Exists*).

Listing 5.4: Symbol Table construction when visiting a method declaration

```
public override void VisitMethodDeclaration(MethodDeclarationContext context)
{
   string methodName = context.methodName().GetText();
   string methodReturnType = context.returnType().GetText();
   SymbolTable.AddSymbol(methodName, methodReturnType, SymbolKind.Method);
   SymbolTable.OpenScope();
   Visit(context.formalParameters());
   Visit(context.methodBody());
   SymbolTable.ExitScope();
}
```

Table 5.2: Operations provided by the Symbol Table

OpenScope()	Initialize a new scope in the last added symbol and move the current scope to there
AddSymbol(name, type, kind)	Add a new symbol to the current scope
ExitScope()	Move to the outer scope of the current scope
EnterScope(name)	Move to the scope of a symbol if present in the current scope
Lookup(name)	Search a symbol in the current and outer scopes and return its information
Exists(name)	Check if a symbol name already exists in the current, outer and all inner scopes

The construction of the symbol table takes place from a *ParserVisitor* who performs the specific operations in the appropriate *visit* methods. Listing 5.4 is an example of how a new method is stored in the table. At first, it is necessary to collect additional information about the method, in this case the name of the method and the return type. Then, a new symbol is added with the collected information and a new scope is opened (remember that a scope is opened in the last symbol added, in this case the method symbol). The next order is to visit the child nodes of the method declaration. One possibility would be to invoke the *VisitChildren* method to visit all the nodes below. However, ANTLR visitors allows you to define the path to follow through the parse tree. In this context, it is only necessary to visit the parameters and the body of the method (we can exclude, for example, the list of exceptions that are thrown by the method). In the end, it is important to leave the scope so you can then visit, for example, the next method and open a new scope in the right place.

Let us now turn to the most important part of the symbol table: how the mutation rules will explore the table. As in construction, it is mandatory to enter and exit the scopes in the appropriate places of the parse tree. The only difference is that no symbols will be added and instead of opening a scope, you enter a scope. Thus, a mutation rule will have to perform operations on the symbol table at the same points in

which it was built. For example, whenever it enters a new class, the rule will have to enter the class scope, visit its child nodes, and finally leave the scope. However, a problem arises.

Let's imagine a rule that is only interested in visiting while loops. To define this rule, it would only be necessary to define the *visit* method corresponding to the while loop. But, since we have to define where we need to enter and leave the scopes in order to explore the symbol table correctly, we end up having to define all the *visit* methods related to scopes (for example, class and method declaration). The solution to this problem was to define a *ParserVisitor* only for exploring the symbol table, i.e. opening and closing scopes. When a rule wants to explore the symbol table, instead of defining a new *ParserVisitor*, it will extend the one in which the exploration is already defined. This means that, using the same example above, the rule will only have to define the *visit* method for the while loop context. Obviously, if it defines a *visit* method in which a scope in the symbol table exploration is opened and closed, the rule will also have to open and close the scope to keep the exploration correct. This means that, when visiting the while loop, the rule will have to open its scope, check if it can apply the transformation, and finally exit the scope.

5.5 Vulnerability Flow

A mutation rule is designed to detect all locations in a sample code where they can apply a certain transformation. However, not all locations may be interesting for our purpose as they may not have an impact on the vulnerability present in the program. This tool is intended to test a security testing application, so transformations need to vary the presence of a security flaw in the code. For this reason, this section introduces the vulnerability flow and describes how this additional information will help the tool in the diversity of vulnerabilities in the code samples produced.

First, let us recall what was explained in chapter 2. The flow of a vulnerability consists of a path of code elements by which the vulnerability manifests itself. For example, in a SQL injection, the flow shows the steps from the user's input to the SQL query that can be exploited by the attacker to affect the database. It is along this path that the tool intends to apply transformations with the greatest impact on the vulnerability.

To obtain this additional information, the tool must receive, as input, an XML file with the vulnerability flow for each sample of code. This file should follow the structure indicated in listing 5.5. This organization has been defined to match the structure of the XML file that can be exported for each result obtained in the Checkmarx SAST application, which is the target that the VSG tool wants to test.

Looking at this structure, it is in the *Path* tag that we find a set of *PathNodes* that we are interested in. Each *PathNode* contains the information for the location of the node in the code sample: file, line, column and text length.

The tool must load this file and store the flow information for each sample code. One small difference is that instead of saving the line, length and node column, the tool will save the corresponding start and stop index in the sample code to easily check if a transformation is contained in the flow. During the application

Listing 5.5: Structure of the XML file with the vulnerability flow

```
<?xml version='1.0' encoding='utf-8'?>
<CxXMLResults>
  <Query>
    <Result>
      <Path>
        <PathNode>
          <Line>22</Line>
          <FileName>C:\sample\Person.java</FileName>
          <Column>38</Column>
          <Length>4</Length>
        </PathNode>
        <PathNode>
          <Line>25</Line>
          <FileName>C:\sample\Person.java</FileName>
          <Column>10</Column>
          <Length>4</Length>
        </PathNode>
      </Path>
    </Result>
  </Query>
</CxXMLResults>
```

of the transformations, after a rule detects all the transformations that can be applied to a sample of code, the tool will have to check and apply only those that affect the flow nodes of the vulnerability. All the rest will be discarded.

The next question is: what is the effect on the vulnerability flow after a transformation is applied? When we change a piece of code, not only are we interfering with one of the flow nodes, but we are displacing the location of the remaining nodes. This means that the information about the location of the nodes becomes outdated.

To solve this situation, the tool will have to update the vulnerability flow after applying a transformation. First, it will add a new node with all the new code that has been added to the program. This will cause all the nodes that are contained in the location where the transformation was applied to disappear and be replaced with a new node with the code inserted. Second, the tool will update the location of all the nodes within the same file that are after the new added node, as these will be the ones affected with the transformation. For this, it is enough to know the positions of the replaced code and the size of the inserted code to repair the locations of the remaining nodes.

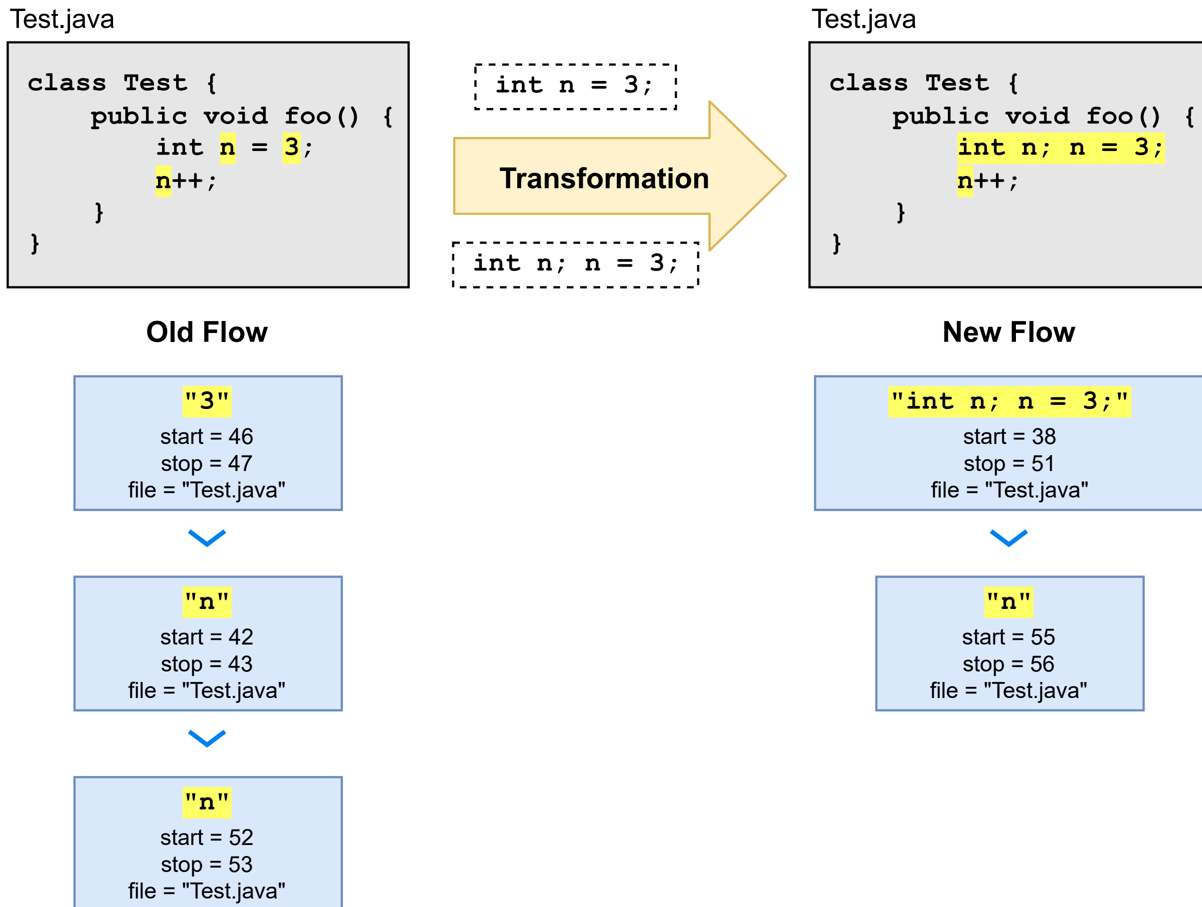


Figure 5.6: Example of how the vulnerability flow will be updated after applying a transformation

Figure 5.6 presents an example of the process of updating the vulnerability flow after applying a simple transformation. When replacing that specific piece of code, the "3" and "n" (index = 42) nodes, that were contained in the transformation, were replaced by the new "int n; n = 3;" node containing all the new code. In the remaining node "n"(index = 52), which was after the location of the transformation, only its location was updated, i.e., its start and stop index went to 55 and 56, respectively.

With all this said, the vulnerability flow presents itself as useful information for the choice of the locations in the program in which the syntactic transformations will have the most impact on the presence of the vulnerability and therefore limits the number of code samples produced that would be useless for the evaluation of the security testing tool.

5.6 Code Transformation

Finally, we arrive at the most important and final part of this chapter. It is in this section that we really find the core of this dissertation and the final step to reach the main goal. All the components that have been described so far, represent the support for the creation and functioning of the mutation rules. It will

be through the application of syntactic transformations in the code samples that it will be possible to test the detection of vulnerabilities in a security testing application.

This section is organized as follows: first, it explains how a mutation rule is implemented to accomplish what is intended; second, it describes how the transformations will be applied singularly in a code sample and, third, how the rules will be combined in order to generate the most diverse samples of vulnerable code. Finally, it will show the set of mutation rules that were implemented for the final analysis of the tool.

5.6.1 Implementing a Mutation Rule

The main function of a mutation rule is divided into two parts. The first part is concerned with detecting all the places in the code where the transformation can be applied and how it will be applied. The second part is related to performing the transformation in the code itself. This section focuses on the first part to describe the process of creating a rule.

Let's start with an example. Consider that the goal is to create a mutation rule that separates the declaration of a new variable from its assignment. That is, if part of the code contains, for example, `int n = 3;` will then be replaced by `int n; n = 3;`. At this point, there's no need to worry about whether the transformation will impact the flow of vulnerability or not.

5.6.1.1 Creating a new Visitor

The first step in implementing a mutation rule is to define a *ParserVisitor*. This can be achieved by extending the one created for the exploration of the symbol table. As with the construction of the symbol table, the *ParserVisitor* will visit the appropriate contexts to obtain the information needed for the transformation. For this case, it will not be necessary to resort to the symbol table since the transformation does not need information about the symbols to be applied correctly.

Let's call the rule we want to implement *SeparateVariableDeclarationFromAssign* and start by creating a new *Visitor*. It will have two instance variables: the name of the file in which the rule is looking for mutations, and a list that will be populated with the mutations found. It will also have a Visit method for the root context of every parse tree (*CompilationUnitContext*), where the list of mutations is initialized and, after visiting all children contexts and collecting all mutations, it will be returned.

The next step is to identify the context or contexts we want to visit in the program structure to assess whether the transformation can be applied and, if it can, to collect the data needed to perform the transformation process. A mutation rule may involve several changes to the contents of a code sample in order to get the final result correct. For example, if the mutation introduces a class that needs to be imported, the rule has to check the list of imported classes and add a new import if the desired class is not present, which means that two local syntactic transformations would be required for the final code not to present compilation problems. Each singular transformation has four characteristics: the name of the

Listing 5.6: Creating the Visitor for the SeparateVariableDeclarationFromAssign rule

```
class SeparateVariableDeclarationFromAssignVisitor : JavaParserVisitor<List<Mutation>>
{
   List<Mutation> Mutations;
   string Filename;

   public override List<Mutation> VisitCompilationUnit(CompilationUnitContext context)
   {
      Mutations = new List<Mutation>();
      VisitChildren(context);
      return Mutations;
   }
}
```

file to which it will be applied, the start and stop indexes that represent the position of the code range that will be removed, and finally the new code to be inserted.

For the example rule, only one context will need to be visited for the transformation, which in the current grammar is called *LocalVariableDeclaration*. In order for the transformation to take place, certain conditions must be met:

- The declaration must be of only one variable (i.e. avoid when there is "`int n = 3, x = 4`")
- The declaration must assign a value to the variable
- The declaration must specify the type of the variable to avoid cases where the `var` keyword is used (introduced in Java 10)

To test these conditions, the *Visitor* presents the `Applicable` method which, receiving the variable declaration context, approves whether the mutation can be performed.

To finish, it is only necessary to gather all the information about the transformation from the context of the variable declaration: the start and stop indexes correspond to the index after the variable name in the declaration (for this specific case, the two indexes will be identical since it is not necessary to remove code), and the new code consists of a semicolon followed by the variable name. So, in the end, all this mutation does is turn "`int x = 1;`" into "`int x; x = 1;`" (the highlighted code represents what was added to the original).

Listing 5.7 shows the final version of the Visitor defined to obtain all possible transformations in a file for the mutation rule to be developed. With this, half the work is done. All that remains is to create the rule that will use this Visitor and apply the transformations to the code.

Listing 5.7: The final result of the Visitor for the SeparateVariableDeclarationFromAssign rule

```
class SeparateVariableDeclarationFromAssignVisitor : JavaParserVisitor<List<Mutation>>
{
   List<Mutation> Mutations;
   string Filename;

   public override List<Mutation> VisitCompilationUnit(CompilationUnitContext context)
   {
      Mutations = new List<Mutation>();
      VisitChildren(context);
      return Mutations;
   }

   public override List<Mutation> VisitLocalVariableDeclaration(
       ↪ LocalVariableDeclarationContext context)
   {
      if (Applicable(context))
      {
         var variableDeclarator = context.variableDeclarators().variableDeclarator()[0];
         string variableId = variableDeclarator.variableDeclaratorId().GetText();

         int start = variableDeclarator.variableDeclaratorId().stop.StopIndex;
         int stop = start;
         string newCode = $"; {variableId}";
         Mutations.Add(new Mutation(Filename, start, stop, newCode));
      }
      return null;
   }

   private bool Applicable(LocalVariableDeclarationContext context)
   {
      return
         // the type of the variable is specified
         context.typeType() != null
         // only one variable is declared
         && context.variableDeclarators().variableDeclarator().Length == 1
         // a value is assigned to the variable
         && context.variableDeclarators().variableDeclarator()[0].variableInitializer() !=
             ↪ null;
   }
}
```

Listing 5.8: The definition of the *FindAllMutations* method for the Rule class. The *Visitor* variable is the visitor created for the rule.

```
public List<Mutation> FindAllMutations() {
   List<Mutation> mutations = new List<Mutation>();

   foreach (var file in Sample.Files)
   {
      var parser = new ParserBuilder().Parse(file.Value);
      var typeInferecenceVisitor = new BuildSTVisitor();
      typeInferecenceVisitor.Visit(parser);

      Visitor.Filename = file.Key;
      Visitor.SymbolTable = typeInferecenceVisitor.SymbolTable;

      List<Mutation> fileMutations = Visitor.VisitCompilationUnit(parser);

      fileMutations = Sample.Flow.PathContains(fileMutations);
      mutations.AddRange(fileMutations);
   }

   return mutations;
}
```

5.6.1.2 Creating a new Rule

After creating the Visitor that will collect all the mutations present in a code sample, the next step is to create the rule itself. The goal is to take the information about the mutations and apply it to the source code. For this, there is a class called *Rule* that will serve as the basis for all rules to be created. The purpose of this class is to define the basic functionality that will traverse all the mutation rules that are created.

Each rule will be defined by two features: the name of the rule and the Visitor used to search for all mutations present in a code sample. Furthermore, every rule will have two methods: the *FindAllMutations()* method (listing 5.8) and the *Apply()* method (listing 5.9).

The *FindAllMutations* method, as the name implies, will search for all mutations present in a code sample. It is at this stage that the Visitors, explained and built in the previous chapters, will finally come into play. For each file present in the sample, the same process will be performed: parsing the contents of the file to get the parsing tree, visiting the tree to build the symbol table, visiting the tree with the Visitor defined for the rule to collect the list of mutations, and finally filtering only the mutations that contain transformations with impact on the vulnerability flow present in the code sample.

As for the *Apply* method, the goal is to apply a specific collected mutation to the code sample. To achieve this, it is necessary to go through the stack of mutation transformations (keep in mind that the order in which transformations are applied is important) and, depending on their information, remove the portion of the target code in the indicated file and replace it with the new code. To finish, it is important to

Listing 5.9: The definition of the Apply method for the Rule class

```
public CodeSample Apply(Mutation mutation)
{
   CodeSample newSample = new CodeSample(Sample.Files, Sample.Flow?.Clone());
   Stack<Transformation> transformations = mutation.Transformations;

   while (transformations.Count > 0)
   {
      var transformation = transformations.Pop();

      if (Sample.Files.ContainsKey(transformation.Filename))
      {
         StringBuilder code = new StringBuilder(newSample.Files[transformation.Filename]);
         int start = transformation.Start;
         int stop = transformation.Stop;
         int length = stop - start;

         code.Remove(start, length);
         code.Insert(start, transformation.NewCode);

         newSample.Flow.UpdatePath(transformation.Filename, start, stop, transformation.
             ↪ NewCode.Length);
         newSample.Files[transformation.Filename] = code.ToString();
      }
   }

   return newSample;
}
```

update the vulnerability flow according to the transformation information so that all subsequent mutation rules know the correct flow locations in the new code sample.

To create the example rule that separates the declaration of a variable from its assignment, all one needs to do is create a *Rule* instance with the name of the rule and the Visitor that was built in the previous chapter 5.6.1.1.

Listing 5.10: Creating the SeparateVariableDeclarationFromAssign rule

```
new Rule("SeparateVariableDeclarationFromAssign", new
    ↪ SeparateVariableDeclarationFromAssignVisitor())
```

5.6.2 Applying the Mutations and Rules Composition

Finally, after the entire process of creating a mutation rule, the main goal of this thesis is close to being achieved. Only the last step is missing, that is, taking a set of developed rules and actually applying them to a set of code samples. With this, this chapter presents the last component of the VSG tool's internal structure, the *RulesApplier*.

At the beginning of this dissertation, it was stated that one of the essential points for this tool would be that all the generated samples be as diverse as possible to cover as many cases of vulnerable code as possible. Having diversified code samples for the same vulnerability makes it more difficult for security analysis tools to detect them. A simple solution would be to generate a code sample for each mutation detected in each rule. However, following this path, the generated samples would have only one difference from the original sample, and the difficulty of detecting the vulnerability present would be low. Therefore, the best option was to compose the different rules in order to obtain more diversity in the generated samples. So, the last step consists not only in applying the rules, but also in combining them.

Two possible paths were considered for rule composition. The first would be to collect all mutations of all rules directly from the original sample and make combinations with at least one mutation from each rule. Unfortunately, this path presented a problem when two or more mutations from different rules had to be applied in the same zones of the code. When applying the first mutation, the information of the transformations of the following mutations would be wrong and the generated code would be syntactically wrong. One way to avoid this would be to separately apply each of the mutations that present conflicts. However, not only would the number of generated samples be low, but also there would not be much diversity between the samples. Therefore, this was not the chosen solution.

The second way to compose the rules is similar to the previous solution, however, the rules are applied by phases in each original sample. According to a predefined list of rules, in each phase, one of the rules will be applied to the samples generated by the previous rule. This means that mutations will be collected from the code modified by one of the mutations of the rule that was applied before. In this way, all mutations are applied to the same code sample from which they were collected, so there is no conflict with mutations from previous rules. At the end of the process, if all rules are applicable, all generated samples will have one mutation from each rule and all will be different. Figure 5.7 shows this procedure for 3 rules from start to finish.

The goal of the *RulesApplier* component is to receive a set of code samples, a list of rules, and to follow the explained process to produce the mutated code samples. While doing so, this component will also update the user with information about the rules applied for each original code sample.

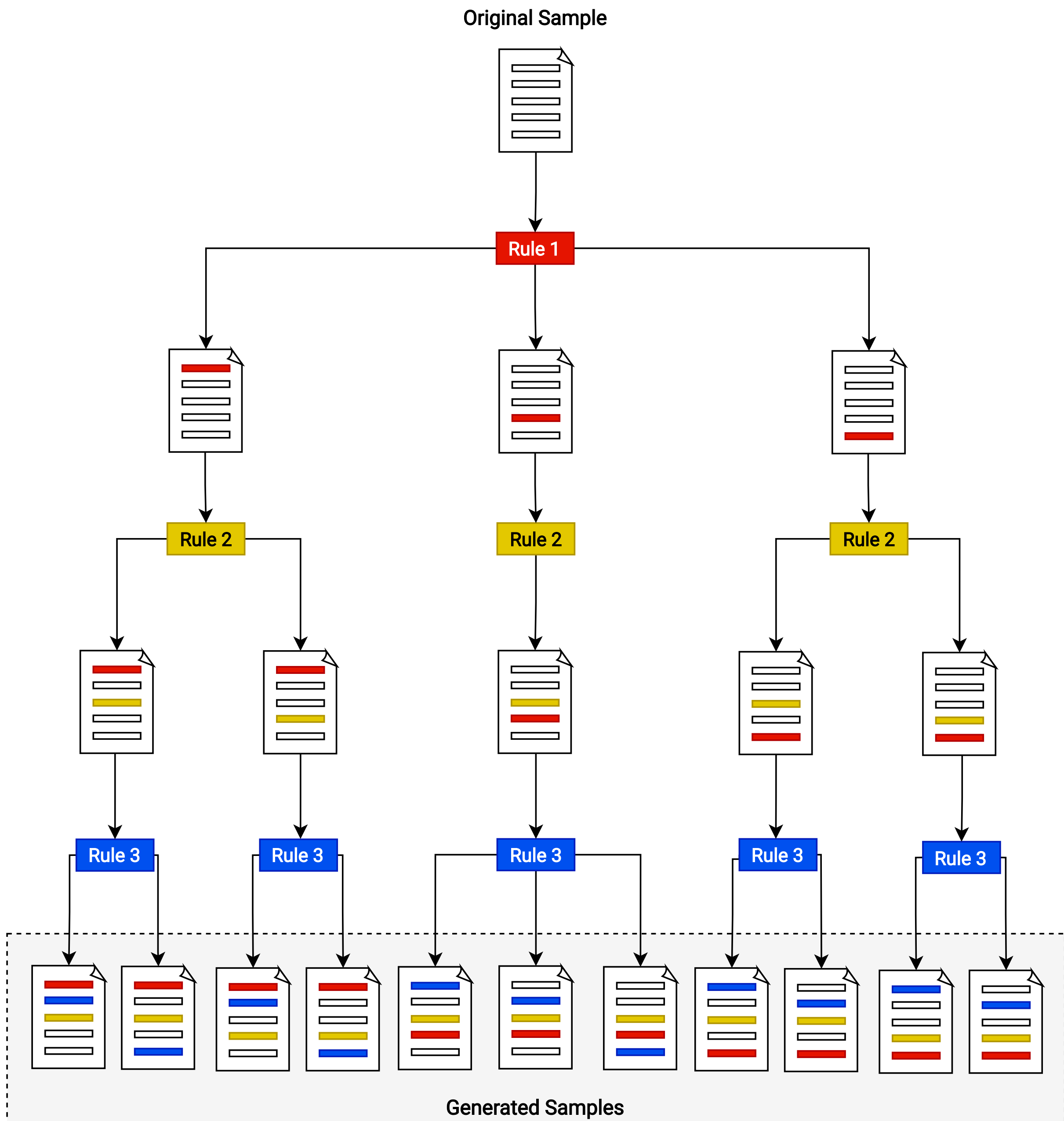


Figure 5.7: Rules Composition Schema. In this example, a list of 3 rules is applied to an original sample generating 11 different versions at the end

Chapter 6

VSG Tests

This chapter is divided in two parts. The first part shows a simple test where a single code sample is used as input to the VSG tool and then compared to all generated samples. This will be interesting to observe where all mutations from different rules are applied in the samples. For the second part, this chapter will meet the main objective of this dissertation: test how VSG can be used to find false negative results in a static application security testing tool which, in this case, will be Checkmarx SAST tool. Since Java is the only language supported, all samples are in Java source code.

6.1 Original code sample vs Generated samples

Before jumping to the differences between the original and generated samples, the following board presents the textual output of VSG.

VSG output during the process

```
Generating samples for vulnerability: Command_Injection
        Sample 'original_sample':
                -> Applying rule ExtractExpressionStatement
                -> Applying rule ReplaceArrayInitialization
Done!
Generated 1 new sample!
```

This shows that two rules were applied to the original sample and only one code sample was generated from the transformations. The first rule, `ExtractExpressionStatement`, is similar to the known refactoring rule ”Extract method”that extracts an expression statement to a new method. The second rule, `ReplaceArrayInitialization`, transforms an array initialization into a list with the same elements that is converted to the respective array.

The listing 6.1 presents the original code sample given as input for this test. It's a simple program that removes and creates files indicated by the user input. Since there is no sanitization for the input, this program has a Command Injection vulnerability, which means the user can execute arbitrary commands on the host operating system. To check the impact of the transformations in the vulnerability, the listing highlights all the flow nodes identified by Checkmarx SAST. It starts on the `args` parameter of the main method and ends in the `exec` invocation of the `Shell` class.

Listing 6.1: Original code sample

```
import java.io.IOException;
import java.util.Arrays;

public class CleanAndCreate {

   static class Shell {
      public void execute(String arg ) throws IOException {
         Runtime.getRuntime(). exec ( arg );
      }
   }

   public static void main(String[] args ) {
      String[] params = { ”rm”, args [1], ”;”, ”touch”, args[2] };
      String res = Arrays.stream( params ).reduce(” ”, String::concat);
      new Shell().execute( res );
   }
}
```

The code sample generated in this test is displayed on listing 6.2. The highlighted green lines represent the differences between the original code sample and, therefore, where the mutations were applied by VSG. The first rule applied was the `ExtractExpressionStatement` rule which, in this case, extracted the `exec` invocation expression to a new method within the `Shell` class. The second rule, `ReplaceArrayInitialization`, transformed how the `params` array is initialized. Instead of a simple initialization with all elements, the array is created from a temporary list to which each element has been added before, including the first element of the `args` parameter which belongs to the flow.

This test, not only shows the transformations applied to a code sample, but also how these mutations affect the presence of a vulnerability, since every rule have an impact in, at least, one element of the flow.

Listing 6.2: Generated code sample

```
import java.io.IOException;
import java.util.Arrays;

public class CleanAndCreate {

   static class Shell {
      public void execute(String arg) throws IOException {
         ffsQXMp(arg);
      }

      private void ffsQXMp(String arg) throws IOException {
        Runtime.getRuntime().exec(arg);
      }
   }

   public static void main(String[] args) {
      String[] params = {};
      List<String> params_temp = new ArrayList<String>();
      params_temp.add("rm");
      params_temp.add(args[1]);
      params_temp.add(";");
      params_temp.add("touch");
      params_temp.add(args[2]);
      params = params_temp.toArray(params);
      String res = Arrays.stream(params).reduce(" ", String::concat);
      new Shell().execute(res);
   }
}
```

6.2 Testing VSG against Checkmarx SAST

Following the main goal of this dissertation, a set of code samples, each with an identified vulnerability, were used as input to test the resulting VSG tool against Checkmarx SAST. Not only different samples were tested, but also multiple kinds of vulnerabilities were used to check if there were any differences in terms of false negative results detected. All vulnerabilities considered of high risk in Java by Checkmarx were chosen to test VSG.

The test was simple. Each initial code sample with a type A vulnerability (detected by Checkmarx SAST) was provided as input to the VSG tool. After this, all generated code samples were analyzed by Checkmarx SAST to detect type A vulnerabilities. If not found, it means that there is a false negative result in the analyzed sample. In other words, for each initial code sample, the number of false negative results detected in the samples generated by the VSG tool was evaluated. Table 6.2 shows the results obtained in the tests.

Using only the 6 mutation rules present in table 6.1, VSG was able to generate more than the initial number of samples provided as input for almost all vulnerabilities. As discussed in chapter 5.6.2, not only

Listing 6.3: Main program of VSG using the list of transformations in table 6.1

```
static void Main(string[] args)
{
   var sampleSet = Load(initialSamplesDirectory);

   var rules = new List<Rule> {
      new Rule("ExtractExpressionStatement", new ExtractExpressionStatementVisitor()),
      new Rule("ExtractIfCondition", new ExtractIfConditionVisitor()),
      new Rule("AddReturnVariable", new AddReturnVariableVisitor()),
      new Rule("StringFormat", new StringFormatVisitor()),
      new Rule("ReplaceExplicitTypeWithVar", new ReplaceExplicitTypeWithVarVisitor()),
      new Rule("ReplaceArrayInitialization", new ReplaceArrayInitializationVisitor()),
   };

   RulesApplier.Apply(sampleSet, rules);

   int generatedSamples = sampleSet.ExportToFiles(generatedSamplesDirectory);
   Console.WriteLine($"Generated {generatedSamples} new samples!");
}
```

is this possible by applying the rules, but also in the combination of applicable mutations each rule has in a code sample.

Table 6.1: Mutation rules used for the test

AddReturnVariable	Change a return expression by declaring a new variable for the return value and returning the variable itself
ExtractIfCondition	Extract the condition of an if statement to a new variable
ReplaceArrayInitialization	Create a new ArrayList using the values of an initialized array and convert it back to an array
ReplaceExplicitTypeWithVar	Whenever possible, replace an explicit type in a variable declaration by the keyword var
StringFormat	Replace a string interpolation expression by a string format expression
ExtractExpressionStatement	Extract an expression statement into a new method

Each of these mutation rules has its own parsing tree visitor from where all the transformations are detected in a code sample. All it's necessary for the whole process is to create a rule from each of these visitors and use them in the *RulesApplier*, together with the initial code samples provided. So, the main program of VSG will have a similar look as in listing 6.3.

Finally, after running VSG and scanning each generated sample with Checkmarx SAST searching for the vulnerability present in the original sample from where it was created, the results in table 6.2 were obtained. The last column of the table shows that false negative results were detected for 6 of the

vulnerabilities used in the initial code samples.

Table 6.2: Results after using Checkmarx SAST to detect in the generated samples the same vulnerability present in the original

Vulnerability	# of initial samples	# of generated samples	# of false negatives detected
Command Injection	10	20	5
Connection String Injection	11	13	0
Deserialization of Untrusted Data	15	7	0
LDAP Injection	10	9	0
Reflected XSS All Clients	70	129	42
Second Order SQL Injection	13	14	1
Stored XSS	15	29	5
XPath Injection	13	20	2
SQL Injection	71	102	20

Chapter 7

Conclusion

This last chapter will be a summary of all the information that was presented throughout this document in order to recall the main goal of the project and how it was achieved.

Any tool that seeks to identify some characteristic in a given object, whether it is detecting a certain protein in a substance, or looking for flaws in production material in a factory, suffers from the possibility of failing its purpose and not being able to find what they are looking for when it exists, which results in false negative results. The same is true for application security testing tools. The problem with these results is that, for the most part, they go unnoticed and do not end up being corrected.

The main idea of static code security analysis tools is to identify vulnerabilities in an application by analyzing the source code. The thesis of this dissertation suggests using code transformation to produce vulnerable code samples that can be used to test the capabilities of these tools. Therefore, this project aims to help discover false negative results and, consequently, failures in the analysis of these security tools.

So, it was necessary to design a tool, which was called VSG, whose function would be to receive code samples and apply transformations that would not interfere with the presence of the vulnerability. In order for the transformations not to be random in the sample body, the vulnerability flow was also introduced as an input to the tool to identify the places where the mutations would be better applied to change the way of detecting the vulnerability.

The next step consisted of using ANTLR to parse the code according to the language grammar. For each mutation rule, there is a Visitor that will traverse the parsing tree generated by ANTLR, and will collect all possible transformations. For some rules, a type inference mechanism was needed, so a symbol table was implemented that works as an auxiliary data structure with the purpose of storing identifiers present in the analyzed code.

After implementing the mutation rules, the last developed component of the VSG tool, *RulesApplier*, will apply the generated transformations to the code sample. In order for the generated samples to be as diverse as possible, the rules were applied in phases. At the end of the process, if all rules apply, all samples will have a mutation of each rule and all will be different.

Finally, VSG was used to test the capability of a security testing tool, in this case Checkmarx SAST. Using 228 vulnerable code samples written in Java as input (along with the vulnerability flows present in each sample previously detected by Checkmarx SAST) and, applying 6 mutation rules, VSG was able to generate a total of 343 vulnerable samples. Of this set, the Checkmarx SAST tool failed to detect the same vulnerability present in the original sample in 75 samples. In other words, 75 false negative results were found. It was then concluded that, through the code transformation process, it is possible to generate samples of vulnerable code that can be used to test the ability of a security analysis tool to detect vulnerabilities.

In the context of the Checkmarx tool, most of the false negative results were identified as a limitation in the flow calculations that will be investigated and, if there is a solution, it will be resolved. The VSG tool not only allowed to discover this flaw, but also provided code samples that will help to solve the problem.

7.1 Future Work

Despite the results obtained, the tool developed within the scope of this project presents not only aspects that can be improved, but also potential features that can be added:

- For this dissertation, only the Java language was supported as an example of the project. However, other programming languages may be supported.
- Instead of applying transformations directly in code, these could be applied at the level of an abstract syntax tree that would later be converted to code again. This could improve the way to implement a mutation rule that, at the moment, presents itself as a laborious process where it is easy to make mistakes.
- The symbol table itself presents several points for improvement.
- Although not related to the main functionality of the tool, the process of identifying false negative results can be automated (scan generated samples with the security tool) so that the final result is only the samples generated in which it was not detected the respective security flaw.

Bibliography

[1] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. *Design of Mutant Operators for the C programming language*. Tech. rep. Purdue University, 1989.

[2] *Application security testing company: Software security testing solutions: Checkmarx*. 2022. url: `https://checkmarx.com/`.

[3] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps. "The care and feeding of wild-caught mutants." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 511–522.

[4] F. W. Calliss. "Problems with automatic restructures." In: *ACM SIGPLAN Notices* 23.3 (1988), pp. 13–21.

[5] R. DeMillo, R. Lipton, and F. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer." In: *Computer* 11 (1978), pp. 34–41.

[6] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. "Towards mutation analysis of Android apps." In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops*. 2015, pp. 1–10.

[7] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei. "Mutation operators for testing Android apps." In: *Information and Software Technology* 81 (2016), pp. 154–168.

[8] A. Derezinska and K. Kowalski. "Object-Oriented Mutation Applied in Common Intermediate Language Programs Originated from C#." In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 2011, pp. 342–350.

[9] M. Endsley. "Automation and situation awareness." In: *Human factors in transportation. Automation and human performance: Theory and applications*. Ed. by R. Parasuraman and M. Mouloua. Lawrence Erlbaum Associates, Inc., 1996, pp. 163–181.

[10] F. C. Ferrari, J. C. Maldonado, and A. Rashid. "Mutation Testing for Aspect-Oriented Programs." In: *2008 1st International Conference on Software Testing, Verification, and Validation*. 2008, pp. 52–61.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[12] B. J. Garvin and M. B. Cohen. "Feature Interaction Faults Revisited: An Exploratory Study." In: *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. 2011, pp. 90–99.

[13] J. Hu, N. Li, and J. Offutt. "An Analysis of OO Mutation Operators." In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 2011, pp. 334–341.

[14] A. Kaur and M. Kaur. "Analysis of Code Refactoring Impact on Software Quality." In: *MATEC Web of Conferences* 57 (2016).

[15] S. Kaur and E. H. Kaur. "Review on Identification and Refactoring of Bad Smells using Eclipse." In: *International Journal For Technological Research In Engineering* 2 (2015).

[16] S. Kim, J. Clark, and J. McDermid. "The Rigorous Generation of Java Mutation Operators Using HAZOP." In: *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications* (1999).

[17] K. N. King and A. J. Offutt. "A fortran language system for mutation-based software testing." In: *Software: Practice and Experience* 21.7 (1991), pp. 685–718.

[18] B. P. Lientz and B. E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.

[19] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. "Enabling Mutation Testing for Android Apps." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 233–244.

[20] R. J. Lipton and F. Sayward. "The status of research on program mutation." In: 1978.

[21] 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)*. 2017, pp. 97–102.

[22] R. Lämmel. "Towards Generic Refactoring." In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming*. RULE '02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 15–28.

[23] Y. Maezawa, K. Nishiura, H. Washizaki, and S. Honiden. "Validating Ajax Applications Using a Delay-Based Mutation Technique." In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. New York, NY, USA: ACM, 2014, pp. 491–502.

[24] E. Mealy and P. Strooper. "Evaluating software refactoring tool support." In: *Australian Software Engineering Conference (ASWEC'06)*. 2006, 10 pp.–340.

[25] T. Mens. "Practical analysis for refactoring." Doctoral dissertation. University of Illinois at Urbana-Champaign, Champaign, IL, 1999.

[26] T. Mens, S. Demeyer, and D. Janssens. "Formalising Behaviour Preserving Program Transformations." In: *Proc. Int'l Symp. Principles of Software Evolution*. 2002, pp. 286–301.

[27] T. Mens and T. Tourwé. "A survey of software refactoring." In: *IEEE Transactions on Software Engineering* 30.2 (2004).

[28] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. "Efficient JavaScript Mutation Testing." In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 74–83.

[29] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. "Guided Mutation Testing for JavaScript Web Applications." In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 429–444.

[30] L. Morell. "A Theory of Error-based Testing." Doctoral dissertation. Univ. of Maryland at College Park, 1984.

[31] E. Murphy-hill and A. P. Black. "Refactoring Tools: Fitness for Purpose." In: *IEEE Computer Society* (2008).

[32] J. Nanavati, F. Wu, M. Harman, Y. Jia, and J. Krinke. "Mutation testing of memory-related operators." In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2015, pp. 1–10.

[33] M. O'Cinnéide and P. Nixon. *Composite Refactorings for Java Programs*. Tech. rep. Dept. of Computer Science, Univ. College Dublin, 2000.

[34] A. J. Offutt. "The Coupling Effect: Fact or Fiction?" In: *SIGSOFT Softw. Eng. Notes* 14.8 (1989), pp. 131–140.

[35] A. J. Offutt. "Investigations of the Software Testing Coupling Effect." In: *ACM Trans. Softw. Eng. Methodol.* 1.1 (1992), pp. 5–20.

[36] A. J. Offutt and R. H. Untch. "Mutation 2000: Uniting the Orthogonal." In: *Mutation Testing for the New Century*. 2001, pp. 34–44.

[37] W. F. Opdyke. "Refactoring Object-Oriented Frameworks." Doctoral dissertation. University of Illinois at Urbana-Champaign, 1992.

[38] W. F. Opdyke and R. E. Johnson. "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems." In: *Proceedings of SOOPPA '90: Symposium on Object-Oriented Programming Emphasizing Pratical Applications*. 1990.

[39] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. "Mutation Testing Advances: An Analysis and Survey." In: *Advances in Computers* 112 (2019), pp. 275–378.

[40] T. Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013.

[41] D. B. Roberts and R. Johnson. "A formal foundation for object-oriented software evolution." Doctoral dissertation. Department of Computer Science, Vrije Universiteit Brussel, 1999.

[42] H. Shahriar and M. Zulkernine. "Mutation-Based Testing of Buffer Overflow Vulnerabilities." In: *2008 32nd Annual IEEE International Computer Software and Applications Conference*. 2008, pp. 979–984.

[43] H. Shahriar and M. Zulkernine. "Mutation-Based Testing of Format String Bugs." In: *2008 11th IEEE High Assurance Systems Engineering Symposium*. 2008, pp. 229–238.

[44] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. "A meta-model for language-independent refactoring." In: *Proceedings International Symposium on Principles of Software Evolution*. 2000, pp. 154–164.

[45] T. Tourwé and T. Mens. "Identifying refactoring opportunities using logic meta programming." In: *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings.* 2003, pp. 91–100.

[46] M. P. Usaola, G. Rojas, I. Rodríguez, and S. Hernández. "An Architecture for the Development of Mutation Operators." In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops*. 2017, pp. 143–148.

[47] P. Vilela, M. Machado, and W. E. Wong. "Testing for security vulnerabilities in software." In: *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications* (2002), pp. 460–465.

[48] M. Ward and K. Bennett. "Formal Methods to Aid the Evolution of Software." In: *International Journal of Software Engineering and Knowledge Engineering* (1995), pp. 25–47.

[49] F. Wu, J. Nanavati, M. Harman, Y. Jia, and J. Krinke. "Memory mutation testing." In: *Information and Software Technology* 81 (2017), pp. 97–111.