MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

Mining and Generating Vulnerable Patterns for Security Testing

Ibragimov, Anton

Award date:
2018

Awarding institution:
University of Namur

UNIVERSITÉ DE NAMUR
Faculty of Computer Science
Academic Year 2017–2018

Mining and Generating Vulnerable Patterns for Security Testing

Anton Ibragimov

Internship mentor: Mike Papadakis

Supervisor: _____ (Signed for Release Approval - Study Rules art. 40)
Patrick Heymans

Co-supervisor: Gilles Perrouin

A thesis submitted in the partial fulfillment of the requirements
for the degree of Master of Computer Science at the Université of Namur

Abstract

ANTON, IBRAGIMOV. Mining and Generating Vulnerable Patterns for Security Testing. (Under the direction of Patrick Heymans.)

Genetic programming is an evolutionary technique that improves individuals from a population in order to better fit the user's needs. In our case, we apply this technique to create new vulnerabilities from an existing repository mined from the Linux kernel. A first step was to create token sequences from this repository. Then, our genetic algorithm derives new vulnerable patterns according to a fitness function that rely on pattern frequency over the vulnerable dataset. Our results indicate that genetic programming can indeed make vulnerabilities more robust over generations. Our patterns fall into two categories: generic patterns applicable to a large number of files and ones that can be used on a smaller set.

La programmation génétique est une technique évolutionniste qui optimise les individus d'une population afin de mieux répondre aux besoins de l'utilisateur. Dans notre cas, nous appliquons cette technique pour créer de nouvelles vulnérabilités à partir d'un dépôt existant miné du noyau de Linux. La première étape était de créer une séquence de tokens à partir de ce dépôt. Ensuite, notre algorithme génétique forme de nouvelles vulnérabilités d'après la fonction d'adaptation qui se repose sur la fréquence des patterns par rapport à l'ensemble des données vulnérables. Nos résultats indiquent que la programmation génétique peut en effet rendre les vulnérabilités plus robustes au fil des générations. Nos patterns se répartissent en deux catégories : des patterns génériques applicables à un grand nombre de fichiers et ceux qui peuvent être utilisés sur un ensemble plus petit.

Acknowledgements

The internship at the SnT in Luxembourg has allowed me to discover the world of research and to gain real experience. It has helped me develop my analytical and interpersonal skills. But more importantly, it provided the chance to work with great people.

I would like to thank Professor Patrick Heymans and Gilles Perrouin from the University of Namur for the opportunity they gave me to complete this internship and for their great assistance while writing this Master Thesis. I would also like to thank members of the Serval team and especially my internship promoter Mike Papadakis for his dedication.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Modern software is complex, adaptive and pervades nearly every aspect of everyday lives. This implies that the aftermath of errors introduced during development [1] can lead to serious consequences, such as critical services disruption or security issues.

Software vulnerabilities are the source cause of computer security problems. The protection of computer systems depends on the identification of vulnerabilities in software and thus, the security became a growing concern. To provide a security layer, algorithms must meet certain criteria in implementations and unfortunately, this is not always the case.

The Heartbleed[1] vulnerability is the perfect example concerning cryptographic algorithms. Heartbleed is a vulnerability that affects OpenSSL, an open source implementation of the Secure Socket Layer (SSL) protocol used to provide cryptographic services. Under normal conditions, this weakness permits stealing protected data by the SSL/TLS encryption used to secure the Internet. In fact, a single missing sanity check in its code source turned it into a huge security hole. Actually, attackers could read sensitive information from an estimated 24-55% of the most popular one million websites [2]. This memorable flaw emphasised the fundamental role of the security inside the source code.

Nowadays, most of critical vulnerabilities are found by manual analysis of the code by experts. Therefore, the security analyst is manually reviewing code - a laborious work requiring an advanced knowledge of the system. The difficulty of this complex task creates a demand for tools to help analysts in their daily work. Hence, these tools could potentially prevent human error.

[1]Source: http://heartbleed.com

1.2 Research objectives

Complexity of software systems is not ready to stop increasing. From this assessment, testing and maintenance of software became really important. These two aspects can take up to 80% of the development cost [3]. Subsequently, there is an urgent need to automate these activities.

The objective of this master thesis is to set an automatic technique to identify vulnerable code parts, similar to what vulnerability prediction models do. The technique mines patterns from past defects and based on them, it predicts likely vulnerable code parts on the code under analysis. In short, mined patterns can be used to guide security inspections by pointing out what and where to check, such as code reviews and testing.

1.3 Thesis structure

This thesis is structured as follows. Chapter 2 presents the key concepts of the thesis such as static analysis and evolutionary computation. Chapter 3 presents my research methods and contributions, that is how from a vulnerable dataset of files, it is possible to create vulnerable mutants. Following all the turnkey concepts, chapter 4 talks about results. At first, this part shows all the parameters for the experiment and then, the results in graphic visualisations. Then, chapter 5 gives a quick review on some existing work that can be related directly. It explains briefly alternative ways of analysing the subject and the connection that can be made with this master thesis. At the end, chapter 6 concern directions for further research in this context. The last chapter 7 resumes the work and the significant conclusions to be drawn.

Chapter 2

Background

This chapter presents the general knowledge concerning the main subject and its content is based on Yamaguchi's thesis "Pattern-based vulnerability discovery" [4]. At this stage, it is essential to develop key concepts in order to be able to understand the general view. There are three important concepts to define: code representation, vulnerability and genetic programming.

It is necessary to be able to manipulate a source code automatically. The solution for this requirement consists of manipulating a representation of the code. There are different ways to represent a code and they are presented in the first section. The aim is to show how a source code can be illustrated and processed.

Another imperative definition concerns the concept of vulnerability in software. It is the starting point for this work and therefore, its terminology must be defined. Plus, it is interesting to explain how a software vulnerability is discovered because all work conducted here rely on this concept.

The last developed concept is the genetic programming. The third section introduces the basic idea with examples in order to understand the way it works in broad terms. Moreover, DEAP, the tool that delivers results is presented with an architectural view.

2.1 Representation of the code

Depending on the purpose and the field, there are different kinds of representations of the code. They have been designed in order to analyse and optimise a specific code. This section is based on Yamaguchi's work [5] with the aim of explaining code representation. Here is the classic representations list :

- Abstract syntax tree;
- Control flow graphs;
- Program dependence graphs.

The following sections are based on a simple source code as seen on listing 2.1.

```
1 void foo () {
2         int x = source();
3         if (x < MAX) {
4                 int y = 2 * x;
5                 sink(y);
6         }
7 }
```

Listing 2.1: A source code

2.1.1 Abstract Syntax Tree

The abstract syntax tree is a classic structure used in program analysis and it is the basis for the generation of many other code representations. The tree encodes how statements and expressions are nested to produce programs. In addition, the abstract syntax tree does not represent the concrete syntax chosen to express the program contrarily to the parse tree. It ignores details of program formulation that does not add information on the semantics and plus, the abstract syntax does not contain punctuation symbols because they are already encoded implicitly in the tree structure.

Generally speaking, abstract syntax trees are convenient for simple code transformations and can be useful to identify semantically similar codes. However, the problem in our case is that AST can hardly work based on code snippets and therefore, it is not flexible enough in our context.

Figure 2.1 shows the abstract source tree for listing 2.1. Regarding to its representation, the abstract tree is an ordered tree where inner nodes are operators and leaf nodes are operands.

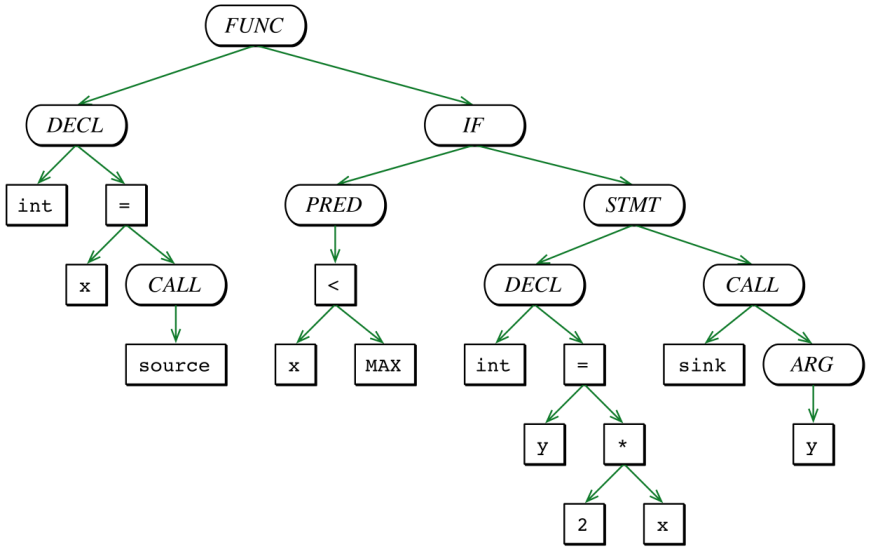

Figure 2.1: The abstract syntax tree

2.1.2 Control Flow Graphs

A control flow graph is an abstract representation of an algorithm where each node represents a basic block of code (a sequence of instructions). The edges in the graph represent possible transfers of control between basic blocks [6]. In other words, a control flow graph describes the code statements order and conditions that need to be respected for a particular path of execution. It is also important to know that edges need to be ordered and labelled (true, false or ϵ). When a block has no outgoing, the edge is labelled with ϵ. Figure 2.2 shows the control flow graph for the listing 2.1.

For example, in the security context, control flow graphs have been chosen to detect malicious applications in Android. As a matter of fact, it should be pointed out that most methods for detection of Android malware are based on permission or on the identification of expert features - unfortunately, these approaches are susceptible to obfuscation techniques. In this case, a method was proposed for malware detection based on control flow graph [7].

Also, control flow graphs have become a standard code representation in reverse engineering to assist in the program understanding. However, even if a control graph flow displays the control flow of a given application, it fails to provide data flow information. To be precise, it means that this kind of graph cannot be efficiently used to pinpoint statements [5].

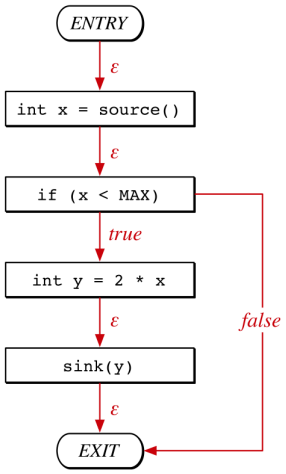

Figure 2.2: The control flow graph

2.1.3 Program Dependence Graphs

The program dependence graph helps to determine all statements and predicates that affect the value of a variable at a specified statement. It represents dependencies among statements and predicates. It has two types of edges that can be seen on figure 2.3.

Data dependency: an edge expressing the influence of one variable on another;

Control dependency: an edge corresponding to the influence of predicates on the values of variables.

The program dependence graph represents the essential data relationships and the essential control relationships without the unnecessary sequencing present in the control flow graph. These dependence relationships determine the required sequencing between operations, showing possible parallelism [8].

In general, an in-depth analysis of the program dependence graph can potentially bring many optimisations. This is firstly due to connections between relevant parts of the program - therefore, many code improvements can be performed thanks to the representation. Plus, the hierarchical part of the program dependence graph allows an effective summarisation of the program.

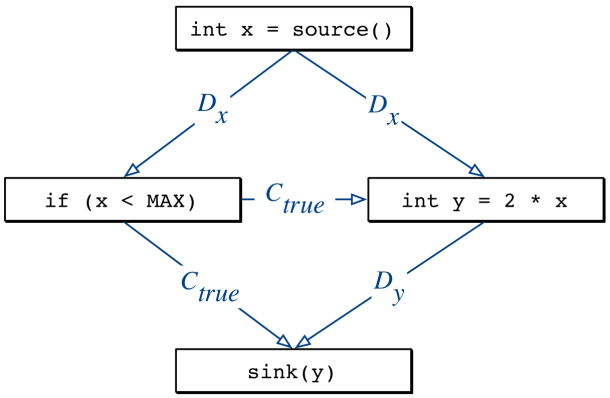

Figure 2.3: The program dependence graph

2.2 Software vulnerability

2.2.1 Terminology

A software vulnerability has one of these three forms [9]:

Access control vulnerability: The operating system performs operations that do not respect the security policy as described by the access control. In other words, there is a need to determine what operations are allowed on system objects for every subject and object in the system.

State-space vulnerability: It is a characterisation of a vulnerable state which tells the difference from all non-vulnerable states [10]. The definition of states dependent on the functionality of the system and thus, each safe and unsafe state must take into consideration the environment of the system. Also, this kind of vulnerability can be defined as unfortunate characteristic that grants a threat to potentially happen [11].

Fuzzy vulnerability: It is a violation of the expectations of users and especially, when the violation is provoked by an external object.

A specification, a development or a configuration error can make software vulnerable. This means that a software vulnerability is an instance of an error.

2.2.2 Vulnerability discovery

This section describes how a vulnerability can be detected and its content is based on Liu's work [12].

To begin with, a software vulnerability can be organised into three different activities :

- Vulnerability discovery;
- Vulnerability analysis;
- Vulnerability exploitation.

The first activity concern software vulnerability discovery that tries to detect existed but unfounded vulnerabilities. Nowadays, discover part has become the focus of researchers. Regarding discovery techniques, it went from manual discovery to computer assisted discovery.

The second activity is software vulnerability analysis. It spotlights on investigate discovered software vulnerabilities. With analysis, it can be possible to detect new or same type of vulnerabilities. Plus, it can help to avoid and defend from this specific kind of vulnerability.

The last activity is related to exploitation. As its name suggests, it concerns a concrete exploitation of the discovered vulnerabilities. It includes the attack using the vulnerabilities and the defence against the vulnerabilities.

Obviously, vulnerability discovery is the basis of the two other categories. In order to discover defects or weaknesses inside the software, there are some techniques:

2.2.2.1 Static analysis

It is the action of checking out a system depending on its content or its documentation without executing the program. In other words, it is about manual or automated (e.g. FindBugs[1]) code inspection. This technique requires spending lots of time and it highly depends on the experience and skills of the analyst.

Static analysis	
Advantages	- No need to execute the program; - Easily integrated into the software development circle; - Able to find bugs before the release of software.
Disadvantages	- High false positive rate; - Need a human to verify the results; - Cannot be entirely automatic; - Unable to detect conception bugs; - Unable to detect vulnerabilities caused by environment.

Table 2.1: Static analysis: advantages and disadvantages

[1] http://findbugs.sourceforge.net/

2.2.2.2 Fuzzing

The concept of fuzzing is characterised by a generation of random characters. In most cases, a lot of generated data is totally invalid. At the present time, researchers proposed new methods of data generation like the data mutation technique. This latter technique requires knowledge about the tested environment and therefore, a human involvement is expected in the process. It is also worth noting that data mutation is more relevant when we have to deal with very complex specifications.

Fuzzing	
Advantages	- Easy understandable; - No false positive; - High automation degree.
Disadvantages	- High randomness; - High false negative; - Low degree of generalisations.

Table 2.2: Fuzzing: advantages and disadvantages

2.2.2.3 Penetration testing

It evaluates the security of a system by reproducing attacks and it determines if these latter were successful. Penetration testing can bring developers a list of vulnerabilities in the environment of test, which can be used to improve security. Note that the penetration testing has become a popular part of quality assurance techniques for web applications [13].

Penetration testing	
Advantages	- No false positive; - Based on practical user environments; - Expose vulnerabilities hard detectable by other tools.
Disadvantages	- Technique heavily depending on human; - Results depend on tester skills and experience; - May hurt the tested system.

Table 2.3: Penetration testing: advantages and disadvantages

2.2.2.4 Vulnerability discovery models

VDMs are probabilistic methods for modelling the discovery of software vulnerabilities. They operate on the operational environment and the vulnerability discovery date. They are a helpful tool for understanding vulnerabilities

and estimating characteristics of software systems [14]. Also, it can predict the future vulnerability discover process.

Vulnerability discovery models	
Advantages	- Use of the discovered vulnerabilities; - Able to predict the rate of vulnerability discovery and the total number of vulnerabilities.
Disadvantages	- Some VDMs base needs to be validated; - Only apply to a single software; - Lack general valid VDMs.

Table 2.4: Vulnerability discovery models: advantages and disadvantages

2.3 Evolutionary Computation

This section describes principles of natural selection applied to the computation, that is the *evolutionary computation*. It is based on Forrest, Koza and Sivanandam works [15, 16, 17].

2.3.1 Genetic terminology

Evolutionary computation is a name wrapping genetic algorithms (GA), genetic programming (GP), evolutionary strategies and evolutionary programming. These concepts are related but they are different [18].

Generally, GAs use a fixed length representation whereas GP uses a variable size structure. Conventionally, the standard representation of each individual in GA is a stream of bits. GP is a variant of GAs where the hypothesis being manipulated are programs rather than bit strings [19]. Plus when it comes to GAs, they do not undoubtedly represent the evolution of algorithms, but rather a subgroup of all possible algorithms.

2.3.2 Basic idea

Genetic algorithms work has begun in the 60s with John Holland at the University of Michigan. In 1975, his first accomplishment was the publication of "Adaptation in Natural and Artificial System". The basic idea is that the genetic pool of a given population potentially contains the right solution, or even a better solution to a problem [17].

In order to understand genetic programming, the connection must be made with the Darwinian natural selection. In nature, biological structures that can

adjust themselves to their environment survive and reproduce at a higher rate over a period of time. In other words, structure is the consequence of fitness.

Hence, a genetic algorithm is a form of evolution. Genetic algorithm is a method that can be used as a problem solver and as a modelling for evolutionary systems. For many types of problems, genetic algorithms can help to evolve a solution with the help of diversified techniques. The latter can include optimisation and determination of the relevant solution. Also, the perfect solution does not exist as these techniques are often used when no optimal deterministic solution can be found.

Generally, genetic algorithms work on simplified representations of the natural world (e.g. binary digits). But they are qualified enough to evolve sophisticated structures. These structures are called individuals (or offspring) and can illustrate a solution form. Note that not every form is a valid solution for the specific problem.

2.3.3 Genetic algorithm example

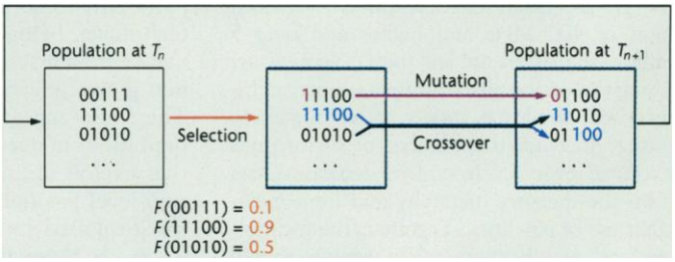

Figure 2.4: Operation of the genetic algorithm

As an example [15], a population of three individuals on figure 2.4 can be seen. At T_n , these individuals are:

- Individual A : 00111
- Individual B : 11100
- Individual C : 01010

Thereafter, a selection is done according to the fitness function. This process can be seen as the individual reordering: the individual with the best fitness score will be on top of the list and the worst one at the bottom. According to figure 2.4, the individual B (11100) has the best fitness score following by the individual C and A. After the selection process, the genetic operators are put into use: the first individual on the list is mutated and the remaining two undergo the crossover process. At $T_n + 1$, there are three new individuals:

- Individual B' : 01100
- Individual AC : 11010
- Individual CA : 01100

Individual B' is the mutation of the individual B. Individual AC and CA are the offspring of the individual A and C.

2.3.4 Individual evaluation

As mentioned before, there is a function called the fitness function. In fact, each individual is evaluated in the environment and he is attached with a numerical evaluation. This latter is defined by the fitness function F as seen on figure 2.4.

The environment can be different depending on the experiment purpose. For example, it could be a computer simulation or an interaction with other individuals. In most cases, fitness function returns a number that will be linked with the individual. The fitness function is the command post for the genetic algorithm if a specific problem has to be resolved.

After having computed all the fitness scores, the algorithm will sort out individuals according to their fitness result. This is when the selection process enters in action: individuals with a low fitness score are eliminated and those with a good fitness score are preserved.

Then, the selected individuals face genetic operators such as the mutation and the crossover in a probabilistic manner in order to deliver a new population of individuals. New generations can be produced in two distinct ways [15] :

1. **Synchronously:** The old generation is completely replaced by the new one;
2. **Asynchronously:** The old and new generation overlap.

Logically, the global fitness score of the population should improve. Therefore, the individuals of the new population are considered as improved solutions to the specific problem.

With the operations of selection, mutation and crossover, the genetic algorithms will gather over generations towards the near global optimum.

2.3.5 Mutation

With the aim of improving a population score, genetic algorithms can perform an asexual recombination to individuals: the mutation operation. Mutation is a way to get new individuals inside a population. It consists in changing

the value of genes. In fact, random changes can be an excellent way of exploring the search space [17].

In genetic algorithms, mutation brings diversity. It operates with only one individual and there are multiple mutation types. The following list presents some operators and it is based on Larranaga's work [20].

Displacement mutation (DM): The DM operator randomly selects sub-list with elements from the core list. This latter is cut and then inserted in a random place.

On figure 2.5, the sub-element (3 4 5) is selected and it is placed after element 7. This kind of mutation is also called "Cut mutation".

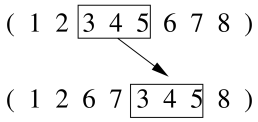

Figure 2.5: Displacement mutation

Exchange mutation (EM): The EM operator randomly selects two elements in the list and exchanges them as it is suggested.

On figure 2.6, the elements 3 and 5 are exchanged. The exchange mutation operator is also called the "Swap mutation operator".

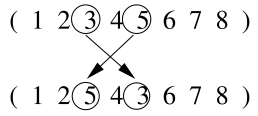

Figure 2.6: Exchange mutation

Insertion Mutation (ISM): The ISM operator choose an element randomly from the list and removes it. Then it inserts the removed element in a randomly selected place.

On figure 2.7, the element 4 is inserted after element 7. The insertion mutation operator is also named the "Position based mutation operator".

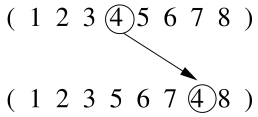

Figure 2.7: Insertion mutation

Simple Inversion Mutation (SIM): The SIM selects two cut points randomly in the list and a reverse operation is applied.

For example, on figure 2.7, the first cut point is between elements 2 and 3. The second cut point is between elements 5 and 6.

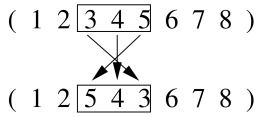

Figure 2.8: Simple inversion mutation

Inversion Mutation (IVM): The IVM operator is comparable to the displacement operator. Likewise, it randomly selects a sub-list from the basic list. Then, this latter is deleted from the list and inserted in a randomly selected position. But this time, the sub-list is inserted in reversed order.

On figure 2.9, the sub-list (3 4 5) is selected, reversed and inserted after element 7. The insertion mutation is also known as the "Cut-inverse mutation operator".

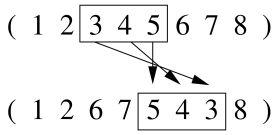

Figure 2.9: Inversion mutation

Scramble Mutation (SM): The SM operator selects a random sub-list and simply scrambles it.

For instance, on figure 2.10, the sub-list (4 5 6 7) is selected and a permutation is applied.

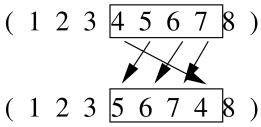

Figure 2.10: Scramble mutation

2.3.6 Crossover

Genetic recombination or sexual reproduction is a key operator for natural evolution. From a technical point of view, it takes two parents and it produces

a new result by mixing characteristics found in the originals. In biology, the most common recombination is the crossover.

The consequence of recombination is decisive because it grants characteristics from two different parents to be assorted. In fact, if the father and the mother possess excellent qualities, it is expected that all the good qualities will be noticeable in the child.

Mutants do not really provide a lot of new fresh individuals. Hence, the crossover operation offers new offspring to the population. There are many crossover techniques and the following list is presented few ones [21].

Partially Mapped Crossover (PMX): The PMX operator attempts to preserve the absolute position of elements. It randomly selects two cut points on both parents. The sub-list in the first parent replaces the corresponding sub-list in the second parent. Then, the inverse replacement is applied outside the cut points with the aim to remove duplicates.

On figure 2.11, the mapped sections are (5 6 4) and (2 3 6). Regarding to the mapping, element 5 becomes 2 ($5 \rightarrow 2$). For the second step, the element 4 becomes 6 and the element 6 becomes 3 ($4 \rightarrow 6 \rightarrow 3$). It can be simplified by saying that element 4 become directly 3 ($4 \rightarrow 3$).

parent 1	:	**1**	**2**	|	**5**	**6**	**4**	|	**3**	**8**	**7**
parent 2	:	1	4	|	2	3	6	|	5	7	8
offspring											
(step 1)	:	1	4		**5**	**6**	**4**		5	7	8
(step 2)	:	1	3		**5**	**6**	**4**		2	7	8

Figure 2.11: Partially mapped crossover

Cycle Crossover (CX): The CX crossover focuses on sub-list that covers the same position in both parents. Then, elements are reproduced from the first parent to the offspring at the same position. The remaining positions are filled with elements of the second parent.

On figure 2.12, (3 6 4) is selected and the cycle is ($3 \rightarrow 4 \rightarrow 6 \rightarrow 3$).

parent 1	:	**1**	**3**	**5**	**6**	**4**	**2**	**8**	**7**
parent 2	:	1	4	2	3	6	5	7	8
offspring	:	1	**3**	2	**6**	**4**	5	7	8

Figure 2.12: Cycle crossover

Modified Crossover: This operator is a simple extension of the one-point crossover. A cut position is taken randomly on the first parent. Then, the offspring is built by appending the second parent part of the initial segment of the first parent before the cut point. Plus, the duplicates are removed.

On figure 2.13, the cut point is between elements 2 and 5 of the first parent. The offspring takes the left part of the cut point and the right part of the second parent. The element 2 is considered as duplicate, so it is replaced.

parent 1	:	**1**	**2**	**5**	**6**	**4**	**3**	**8**	**7**
parent 2	:	1	4	2	3	6	5	7	8
offspring	:	**1**	**2**	4	3	6	5	7	8

Figure 2.13: Modified crossover

Order Crossover (OX): The order crossover attempts to preserve the relative order rather than the absolute position. The OX operator extends the modified crossover by establishing two cut points randomly. The sub-list between the two cut points in the first parent is copied to the offspring. Then, the remaining positions are completed depending on the elements of the second parent from the second cut position.

On figure 2.14, the sub-list (5 6 4) is between the two cut points. The first step takes this sub-list without any modification. The second step takes elements from the second parent at the second cut point and puts them in the offspring - these elements are (5 7 8 1 4 2 3 6). If a element of the last list appears in the sub-list (5 6 4), they are ignored. Therefore, the final list to be added to the offspring is (7 8 1 2 3).

parent 1	:	**1**	**2**	**5**	**6**	**4**	**3**	**8**	**7**
parent 2	:	1	4	2	3	6	5	7	8
offspring									
(step 1)	:	–	–	**5**	**6**	**4**	–	–	–
(step 2)	:	2	3	**5**	**6**	**4**	7	8	1

Figure 2.14: Order crossover

Order Based Crossover (OBX): The OBX operator spotlights the relative order of the parent elements. At first, a set of elements is chosen in the first parent and they are put in the same order in the offspring. But, their positions are chosen depending on the second parent. The remaining positions are completed with the elements of the second parent.

On figure 2.15, elements (5 4 3) are selected in the first parent (the order is important). In the second parent, these elements are found in a different order at position (2 4 6). As a consequence, elements (5 4 3) takes place at position (2 4 6) in the offspring. The remaining positions are completed with elements from the second parent.

parent 1	:	**1**	**2**	**5**	**6**	**4**	**3**	**8**	**7**
parent 2	:	1	4	2	3	6	5	7	8
offspring	:	1	**5**	2	**4**	6	**3**	7	8

Figure 2.15: Order based crossover

Position Based Crossover (PBX): The PBX focuses on the relative order inheritance from the parents. Concerning the absolute position of the elements from the second parent, it is barely kept. Regarding the functioning, a list of positions is selected in the first parent. Then, elements from this position are copied to the offspring at the same position. The other positions are completed with the remaining elements.

On figure 2.16, positions for elements (5 4 3) are selected in the first parent. Later, they are found at the same position in the offspring. The remaining part of the offspring is filled with elements from the second parent.

parent 1	:	**1**	**2**	**5**	**6**	**4**	**3**	**8**	**7**
parent 2	:	1	4	2	3	6	5	7	8
offspring	:	1	2	**5**	6	**4**	**3**	7	8

Figure 2.16: Position based crossover

On another note, in the context of the travelling salesman problem, it is interesting to point out that the order-preserving crossover operators are superior to the operators preserving the absolute position of the elements. In fact, in the experiment, the order crossover (OX) operator is better than the partially mapped crossover (PMX) and the cycle crossover (CX) [22].

2.3.7 DEAP: a novel evolutionary computation framework

The distributed evolutionary algorithms in Python (DEAP [2]) is a novel evolutionary computation framework for prototyping and testing ideas [23] developed at the Laval University. In contrast to common black box type of frameworks, DEAP looks for explicit algorithms and transparent data structures. Here, the term "black box type framework"means that it brings high level functionality and it does not bring implementation details. For instance, some evolutionary computation frameworks like EO [24], ECJ [25] or Open BEAGLE [26] are black box types.

As a matter of fact, these listed frameworks can be compared to DEAP even if they work in a different way. For example, ECJ provides an alternative higher level programming language by offering to set up the algorithms

[2]Source: https://github.com/DEAP/deap

via a pipeline concept submitted to the framework through a configuration file. When it comes to Open BEAGLE, there is a possibility to customise the algorithm parameters via an external XML file. It is clear that from a software engineering point of view, these solutions are fascinating and interesting because they can adapt an algorithm in order to meet the requirements. Nevertheless, the developer returns to the main problem : they are black box frameworks.

In the context of the evolutionary computation, black boxes can become a handicap when it is about exploration for new algorithms or using custom methods. Plus, if documentation is missing, the developer needs to look into the code details at a lower level. Thus, the framework DEAP provide a toolbox that allows to write a custom evolutionary algorithms and every aspect of the process.

It must be emphasised that DEAP is developed in Python programming language. This latter has comprehensible syntax and is accessible for any developer. In addition, it supports the object-oriented programming and has a dynamic runtime environment. However, Python has some a major drawback: the fact that it is interpreted implies a slow execution compared to compiled languages like C. The project was developed in Python in the first instance because it is easier to develop a prototype in this programming language in order to test ideas.

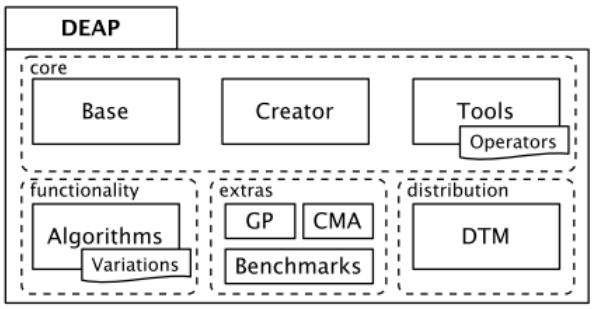

Figure 2.17: DEAP architecture

The architecture of DEAP is assembled around different components as seen on figure 2.17. There are three modules in core section.

1. **Base**: It contains objects and data structures regularly used in evolutionary computation. It has three classes:
   - a basic fitness;
   - a coded tree;
   - a toolbox which is a package for the operators that the developer wants to use in his personal evolutionary computation.

2. **Creator**: It is a meta-factory that offers creation of classes giving freedom to the user from the weight of a class definition. In other words, attributes, data and functions can be dynamically added to create new object classes.

3. **Tools**: It contains regularly used evolutionary computation operators. Moreover, it gives analysis tools such as check-pointing, statistics and genealogy.

Thereafter, there are three other sections which goes with the core section and it has five different modules.

1. **Algorithms**: It contains common algorithms in evolutionary computation. Developers can add their custom algorithms.

2. **GP**: This module contains operators and data structures relating to genetic programming.

3. **CMA**: The covariance matrix adaptation evolution strategies can be found in this module. It is an evolutionary algorithm for difficult non-linear con-convex black-box optimisation.

4. **Benchmarks**: As its name suggests, it provides benchmark functions for analysing algorithm efficiency.

5. **DTM**: It means "Distributed Task Manager" and it manages parallelism.

In conclusion, the DEAP framework merges the adaptability and the potential of the Python programming language with a complete evolutionary computation components. These two aspects promote a rapid prototyping of new ideas and strengthen developing custom algorithms.

Chapter 3

Research Method and Contributions

3.1 Vulnerable dataset

Once a vulnerability is discovered, the analysis part can start. This section explains in detail the vulnerable dataset that will be used as the basis work and its content relies on paper published by the Serval team from SnT [27, 28].

3.1.1 Origin of the dataset

Effective research should be linked to a specific environment since that a global solution fitting on every domain does not exist. In this case, the vulnerable dataset is based on a specific project, the Linux kernel. In fact, studies proved that cross projects are commonly less effective than the project specific ones [29].

The reason for choosing the Linux kernel as a starting point is due to the fact that its community is well organised and works with strict instructions. Consequently, the Linux kernel accumulates a well-reported history of vulnerabilities and therefore, it is easy to gather all the useful information. Moreover, the fact that the Linux kernel is open-source flavours experimentation and research. Also, it is important to mention that the Linux community created the git platform and they were the first to use it in 2005. Thus, from that point of view, the stability of the version control system implies a strong basis for a decent database.

The vulnerable dataset should be supported by a strong foundation. To achieve it, a mining is performed on all Linux kernel vulnerabilities reported in the National Vulnerability Database (NVD) between 2005 and 2016. As a

reminder, the NVD[1] is the U.S government repository of standards-based vulnerability management data. It includes databases of security checklist references, security-related software flaws, misconfigurations, product names, and impact metrics.

To assemble the vulnerable dataset, the Common Vulnerabilities and Exposures (CVE) database is employed. Here, a vulnerability is considered as a program element that was modified to fix vulnerabilities. Furthermore, the CVE database is chosen because it is among the largest vulnerability databases and it includes vulnerabilities that are recognised by the Linux developers.

3.1.2 Building the dataset

The vulnerable dataset will contain well-known vulnerabilities. Before achieving it, a process of collecting vulnerable files is set up. As mentioned before, the core basis is the CVE-NVD database concerning the Linux kernel. The process follows these steps:

1. Collecting all remote repository git URLs of the Linux kernel;
2. Designing a regular expression for extracting the commit hash from the git URLs that mention these repositories;
3. Extraction of all commit hashes that are in the URL of the vulnerability reports using the regular expression;
4. Finding all the commits with a CVE number reference;
5. Retrieving all vulnerable files listed by the commits mentioned before.

However, some vulnerability occasionally affects parts in the Linux kernel written in assembly code. The choice of only considering code written in C is motivated by the fact that the biggest part of this open-source project is written in this programming language.

On another note, this process makes a hypothesis that the known and fixed vulnerabilities are accurate. But eventually, some of this data may represent some false negatives or false positives. Nevertheless, taking into account the history of the Linux kernel vulnerability report, there seems little likelihood of such scenarios.

[1]Source: https://nvd.nist.gov

3.1.3 Usage of the dataset

Once the vulnerable dataset is generated with the collector, the next concerns the retrieval of the useful information. For each vulnerable file from the dataset, an extraction of two different files is performed. More specifically, here is the list of the retrieved information:

- The vulnerable file name and its content.
- The patched of the same file and its content.

To sum up, the information concerns the same file but with a different content that is examined. In practice, the content variation between the two files is the vulnerability indicator.

To highlight differences between two files like in git, the GNU Diffutils package[2] is chosen. Through this package, the "diff" command is used for showing differences between two files. Plus, the unified format is selected for the output. In other words, the output format is more condensed because it ignores superfluous context lines. The command line for listing 3.1 shows a basic diff command.

```
1 diff -U file_1 file_2 > result.git
```

Listing 3.1: Diff command

- **diff**: the diff command;
- **-U**: the unified format;
- $file_1$: the vulnerable file;
- $file_2$: the patched file;
- **result.git**: the destination file with the output result.

On output, a list of chunks is generated depending on the modifications. Each chunk starts with a line that looks like on listing 3.2.

```
1 @@ −N_1 ,N_2 +N_3 ,N_4 @@
```

Listing 3.2: Diff output

- N_1: The file line number where content was removed;
- N_2: The number of lines that were removed;
- N_3: The file line number where content was added;
- N_4: The number of lines that were added.

To illustrate a concrete result, listing 3.3 shows a part of an output. This actual chunk is from the file "net_namespace.c" that was found in the vulnerable dataset. Lines starting with a minus (-) are the lines that were removed. With the same logic, lines starting with a plus (+) are the lines that were added.

[2]https://www.gnu.org/software/diffutils/

Clearly, following the logic of a normal software development, the buggy part of the source code is removed with a security patch. In other words, the removed lines can be considered without any doubt as vulnerabilities. Because of that, the next step implies selecting these removed lines and treat them as vulnerabilities.

```
@@ -317,7 +353,7 @@

        list_add_tail(&ops->list, list);
-       if (ops->init) {
+       if (ops->init || (ops->id && ops->size)) {
                for_each_net(net) {
-                       error = ops->init(net);
+                       error = ops_init(ops, net);
                        if (error)
                                goto out_undo;
```

Listing 3.3: Git result

3.1.4 Dataset technical management

From the technical point view, a Matthieu Jimenez[3] project is used in order to generate a vulnerable dataset object. From this point, a Java program is developed to handle the content. To summarise, here are the technical steps for this section:

1. Extracting the vulnerable dataset from a previous work;
2. Selecting all the vulnerable and patched files stored in the dataset;
3. For each vulnerable and patched file where the file name is the same, the "diff" command is executed and the result is saved in a new file;
4. The result is read with a Java program in order to collect information;
5. A regular expression is applied to save useful information for each git result: the line numbers, the added/removed lines and the file name.

From there, the vulnerable content is now ready to be processed. The next phase is concerning the lexical analysis as the vulnerable part of the dataset is now extracted and kept under a structured program.

With the modelling point of view, an output like on listing 3.3 is processed with a personal architecture that can be seen on figure 3.1. There are three different classes: GitFile, GitChange and GitLine.

As its name indicates, GitFile represents a ".git" file. It contains information about the file name and the location. A GitFile object possesses changes for a given file - this is the concept of GitChange.

[3]https://github.com/electricalwind

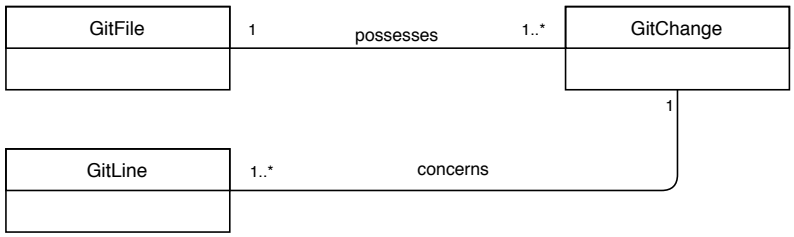

Figure 3.1: Git object diagram

For instance, each chunk delimited by the line seen on listing 3.2 represents a GitChange object. Each object of this type has information about concerned lines. When it comes to the content, the idea of GitLine comes up. In fact, each GitChange has one or multiple line altered. On listing 3.2, there are two changes than begin on lines 4 and 7.

The concept of GitLine is to couple each modifies part. For example, the two changes mentioned before represents two GitLine objects where the modified content can be found.

The presented structure facilitates the lexical analysis. In fact, this architecture points out directly the useful information like the file name and the changed content. Each removed content is considered as vulnerable part and therefore, the lexical analysis focuses on this element.

3.2 Lexical analysis

3.2.1 Motivation

There is a lot of scientific work based abstract syntax tree. As the name suggests, the representation is based on the concept of a tree as a whole. The problem is that vulnerable code is often given in terms of snippets which complicate the task of parsing it using AST tools. Unlike the syntax tree, this technique is based on a sequence of tokens.

3.2.2 Tokenization

In the world of programming languages, words are objects. These objects can be variable names, numbers, keywords, identifiers, etc. Generally, "token" is the name given to these objects. In this case, a token is a categorised block of text. It is frequently defined by regular expressions, which are treated by a lexical analyser.

A lexical analyser will take in input a character sequence and will divide into tokens. Writing a lexical analyser is not very difficult. In fact, the process

needs to read an input and sees if the encountered token is a keyword, an identifier, a whitespace, etc [30]. In short, the process of dividing the input text into units called tokens is referred to as tokenization [31].

3.2.2.1 C-tokenizer : the external tool

To begin with, there is a tool named "C-tokenizer" [32] and as its name suggests, it is a tokenization tool providing a basic C code analysis. It is based on the "tokenizer" package that can be found with "npm", the package manager for the JavaScript runtime environment node.js.

For the technical part, the tokenizer interface follows the WriteStream from node.js. The tokenizer is looking for the longest string in order to match it with one or more rules. For the record, rules are regular expressions associated with a type name. Listing 3.4 shows a source code that will be tokenized with the node command. The result is shown on table 3.1.

```
1 #include "stdio.h"
2 #include "stdlib.h"
3 /* Ex1 */
4 int main(int argc, char **argv) {
5         printf("%d\n", foo(atoi(argv[1])));
6         return 0;
7 }
```

Listing 3.4: A source code to be tokenized

directive =>"#include"	operator =>","	identifier =>"atoi"
whitespace =>" "	whitespace =>" "	open paren =>"("
quote =>"\"stdio.h"\""	identifier =>"char"	identifier =>"argv"
whitespace =>"\n"	whitespace =>" "	open square =>"["
directive =>"#include"	operator =>"**"	number =>"1"
whitespace =>" "	identifier =>"argv"	close square =>"]"
quote =>"stdlib.h"	close paren =>")"	close paren =>")"
whitespace =>"\n"	whitespace =>" "	close paren =>")"
area comment =>"/* Ex1 */"	open curly =>"{"	close paren =>")"
whitespace =>"\n"	whitespace =>"\n \t"	operator =>";"
identifier =>"int"	identifier =>"printf"	whitespace =>"\n\t"
whitespace =>" "	open paren=>"("	identifier =>"return"
identifier =>"main"	quote =>\"%d\\n""	whitespace =>" "
open paren =>"("	operator =>","	number =>"0"
identifier =>"int"	whitespace =>" "	operator =>";"
whitespace =>" "	identifier =>"foo"	whitespace =>"\n"
identifier =>"argc"	open paren =>"("	close curly =>"}"

Table 3.1: C-Tokenizer output

On output it gives for each token, a string type associated the original source string. There are 19 sorts of token that can be seen on table 3.2.

Type of token		
Area comment	Directive	Open square
Area comment continues	Identifier	Operator
Character	Line comment	Quote
Character continue	Line continue	White space
Close curly	Number	Word
Close parenthesis	Open curly	
Close square	Open parenthesis	

Table 3.2: Basic token output

The number of tokens from the basic tool is inadequate for an in-depth analysis. To remedy this situation, more tokens were added in order to specify more elements from the source code. Therefore, adjustments were made for the operator and the identifier tokens.

3.2.2.2 C-tokenizer : the internal modifications

The "identifier" token is divided in six different categories. Each category has a set of tokens that relate to it. For example, the "Relational" category has tokens like "==", "!=", ">", etc. This specialisation provides 39 new tokens.

- Arithmetic
- Assignment
- Bitwise
- Logical
- Member and pointer
- Relational

Concerning the operator token, it has the same logic and it has eleven new categories that provides 32 tokens. For instance, the "Flow control iteration" category holds tokens "do", "for" and "while".

- Data size
- Data type
- Flow control iteration
- Flow control selection
- Loop control
- Redirection flow
- Return type
- Storage class
- Structure
- Type quantifier
- Qualifier

In addition, there is another added token : the "Joker". The idea is simple, this token represent any token. In other words, it is introduced as wildcard character that can be interpreted as any other token. For example, table 3.3

presents two patterns with the same size where tokens are represented as integers. The last token of Pattern A and the forth token of Pattern B are both Jokers (J). Because of that, the two patterns are considered similar during the experiment.

Pattern A	1	2	3	4	5	6	(J)
Pattern B	1	2	3	(J)	5	6	7

Table 3.3: The Joker

3.2.3 Token application

After having defined the concept of token, the next step relates directly to its usage. In fact, everything is going to revolve around this concept directly or not.

To begin with, vulnerable chunks from the vulnerable dataset are tokenized. Actually, a pattern is set up from these chunks. In other words, a vulnerable pattern is created from the set of tokens representing these vulnerable chunks.

Here, the concept of the pattern relates to a flow of abstract information. As a matter of fact, the information is represented with tokens where the sequence is essential. This sequence is defined by the process of tokenization. Thus, a vulnerable part is transformed into a set of tokens that represent a flow of vulnerable information.

From a technical point of view, a token is linked with a unique identification that represents him in the code. The "enum" type in Java responds to these special needs. This compiler-generated class contains a set of consistent data. Internally, an enum value includes an integer, corresponding to the order which it is declared in the source code. Therefore, a vulnerable chunk is represented by a list of ordered tokens.

To sum up, a vulnerable code is transformed to a set of tokens. These tokens are put inside an ordered list that represents a vulnerable pattern. The purpose is to be able to find the same kind of pattern somewhere else and be capable of providing the likelihood of a potential vulnerability.

3.2.4 Pattern structure representation

The pattern structure implementation is straightforward. On figure 3.2, there are two main concepts: the PatternStructure and the Expression. The idea of an Expression object is to take into account the output of the C-tokenizer presented in section 3.2.2.1. In fact, a closer look on table 3.1 spotlights a pairing between a token name and a source code chunk. Therefore, in an Expression object, there are two attributes: the token and its content.

Considering the fact that the number of recognised token is defined earlier, there is a list that classifies them in the code. To be more specific, when the C-tokenizer produce an output like on table 3.1, the program applies a regular expression in order to separate data. Each token is identified and linked to an unique integer. Then, a token is linked to the part of the code it represents.

Once a set of Expression objects is created from the C-tokenizer for a given vulnerable chunk, they need to be put together. To this end, the PatternStructure object is created. A PatternStructure contains an ordered list of Expression objects and the source file name.

With this type of structure, the source code decomposition is honoured. A vulnerable chuck from a given file is transformed into tokens and classified. Thus, there is no significant loss with this method because the opposite process can be done: a PatternStructure can rewrite the vulnerable code in C from the token set.

For example, table 3.4 presents a pattern structure for a code. The first row is the code and the second one is the corresponding token sequence. Typically, a PatternStructure object represents all the information from this table and an Expression object stores the connection between a token and its content. Also, it is important to note that white spaces are not taken into account and therefore, the size of a pattern is determined without them (on table 3.4, the size is equal to seven).

Figure 3.2: Pattern structure object diagram

int	i	=	foo	(	)	;
Data type	Identifier	Assignment	Identifier	Open Paren	Close Paren	Operator

Table 3.4: A pattern structure example

3.2.5 Data cleaning

The tokenizer produces a lot of useless data. As a consequence, filtering the first output is imperative in order to make the data workable. Thus, an optimising process is implemented.

The filtering process analyses the output and removes noises. The procedure consists in eliminating some specific cases: white spaces and comment areas. These two points implies on filtering four different tokens:

- Area comment;
- Area comment continue;
- Line comment;
- White space.

On listing 3.5, we can clearly see two types of comment: an area and a line. Plus, there is a white space between them. All this to say that these kinds of tokens are useless if a meaningful result is the goal. Thereupon, these tokens are considered as noise and are removed.

```
1 /*
2  *      Some useless comments.
3  */
4
5 // Here is another example.
```

Listing 3.5: Example of useless source code

3.3 Evaluation strategy

Having a population of vulnerable patterns without any precision is not enough. In fact, a good strategy must be taken in order to classify them. An ordered list with a score would help to define the way for further decision-making. Therefore, good vulnerable patterns must be put forward and bad ones must be put away. Hence, a fitness function will come into play.

3.3.1 Fitness function

A fitness function helps to measure the quality of the represented solution. In other words, it defines how good a solution is. Each solution needs to be linked with a score to indicate how close it meets the desired solution. In this case, the function lies at the core of an evolutionary algorithm and clearly affects its efficiency.

The score should help to get an idea of the solution. A score of zero does not really point out if the solution is good or bad. Plus, a fitness function score will note increase indefinitely : the population will improve to a certain point and then it will stabilise. Thereby, the algorithm must stop at a particular moment.

Obviously, a calculation must be done to obtain a fitness score value. This computation takes into account a set of variables that must be specified. Starting from the confusion matrix, this section presents these variables and how they are used at a later stage.

A confusion matrix is a table that contains information about actual and predicted classifications done by a classification system. Performance of a classification system is evaluated using the data in the matrix. The table 3.5 shows the confusion matrix for a two class classifiers [33].

		Actual	
		Positive	**Negative**
Predicted	**Positive**	True positive	False positive
	Negative	False negative	True negative

Table 3.5: The confusion matrix

In practice, the confusion matrix works with three different aspects. To be more specific, they are :

1. The vulnerabilities;
2. The vulnerable files;
3. Random files considered as not vulnerable.

True positive (TP): TP is the number of correct predictions that an instance is positive. In this context, this is the number of vulnerable files where a vulnerability is found.

True negative (TN): TN is the number of correct predictions that an instance is negative. Here, this is the number of non-vulnerable files where a vulnerability is not present.

False positive (FP): FP is the number of incorrect predictions that an instance is positive. Also known as a Type I Error, the error of rejecting a null hypothesis when it is actually true. More simply, it happens when the test assumes the existence of something that is not here. In this experiment, this is the number of non-vulnerable files where a vulnerability is found.

False negative (FN): FN is the number of incorrect predictions that an instance is negative. It is also called a Type II Error, the error of not rejecting a null hypothesis when the alternative one is correct. Here again, this is the number of vulnerable files where a vulnerability is not found.

To sum up, figure 3.3 shows a decision tree where numbers for each variable from the confusion matrix is calculated.

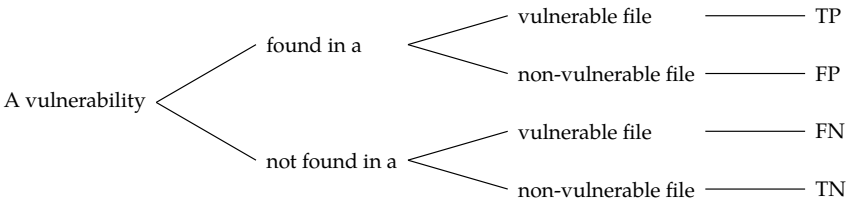

Figure 3.3: Fitness calculation variables

In order to measure the quality of binary classifiers, the Matthews correlation coefficient (MCC) is suitable for the situation. Using the MCC provides a benchmark indicating on how well a classification is performing. In the context of the experiment, the MCC is the fitness formula.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

This formula gives on output a score between -1 and +1. A score of -1 indicates that total opposition between prediction and observation and a score of +1 shows the contrary, i. e. a perfect prediction. In the middle, a score of 0 points out that there is no effect.

To put it simply, if a vulnerable pattern gets a negative score, it means that this pattern is a bad one. In other words, this bad vulnerable pattern has a huge number of false positive and false negative results. That is to say, this bad vulnerable pattern is most often found in non-vulnerable files and not enough found in vulnerable files.

Another detail bears mentioning concerns the lack of results. The fact that the denominator of the formula is a square root implies borderline cases. Hence, the following condition is implicit :

$$(TP + FP)(TP + FN)(TN + FP)(TN + FN) > 0$$

There are therefore circumstances where the MCC returns an error for a division by zero. In the experiment, this kind of event is not blocking because it simply says that the given vulnerability does not produce a tangible result. Consequently, this given vulnerability is skipped.

With this method, it is easy to spotlight good results and remove bad ones. The only problem concerns the performance because this process is time consuming. The efficiency improvement will be discussed later in detail.

3.3.2 Training data sensitivity

In order to set up a predictive model, there are two concepts to take into account: the bias and the variance [34]. Bias is a learner's inclination to gain information the same wrong thing. Variance is a trend to learn random things regardless the real data.

Figure 3.4 shows an example of bias and variance combination. When the bias is high and the variance is low, the learner algorithm is coherent but unreliable on average. On the opposite, when the bias is low and the variance is high, the learner algorithm is accurate on average but unpredictable.

There is a middle-ground between a high bias and a high variance. In the first case, algorithms with a low variance tend to be less complex and have a

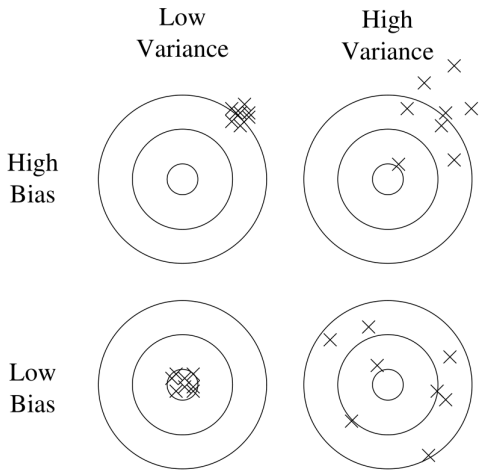

Figure 3.4: Bias and variance

fixed structure. In the second case, algorithms with a low bias turn to be more complex but with a flexible structure.

As a consequence, an algorithm with an elementary structure makes underfit models that cannot gain information from the data. And when it comes to a complicated algorithm, it produces overfit models that collects the noise instead of the useful data.

To get excellent forecasting, a balance must be made between bias and variance. Thus, that would make smaller the error and avoid overfitting and underfitting. Figure 3.5 there are three plots: the first is an undefitting model, the second is a good model and the last one is an overfitting model. The aim is to obtain a model that fits the true function correctly as it can be seen on the central plot. The other two cases are presented in more details after.

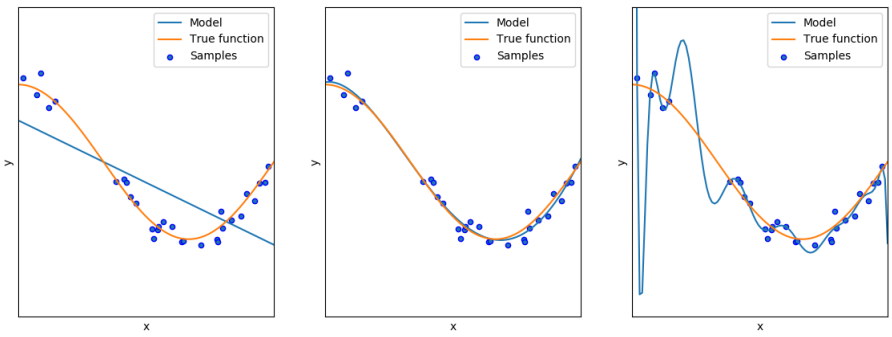

Figure 3.5: Case of underfitting and overfitting

3.3.2.1 Overfitting

A poor performance in machine learning can be explained with the problem of overfitting the data. In a classic way, the aim of a machine learning model is to generalise from the training data to any data from the problem domain. This allows to make prediction on data the model has never encountered.

The overfitting problem indicates that the algorithm knows the training data and it happens when a model learns too much from the latter. Therefore, if the same training data is used over and over again, the results will be skewed. In other words, the fact that the model comes across the same noise and the same variation of data implies that at one point, the model will not be able to generalise the solution efficiently.

On figure 3.5, the right plot shows an example of an overfit model. In fact, the model is too well trained and as a consequence, practically all the samples are taken into account by the model. Also, as it can be seen, the model is distant from the true function at some point.

Solving this issue is uncomplicated in the case of this master thesis: each time the algorithm needs to perform a test, it must take a new sample from the training set. It is clear that the training set must be big enough to deal with this problem. Moreover, the algorithm must take into account that a too long process of testing can have a negative impact on the results. In the same spirit, the model will learn the irrelevant detail and noise in the training dataset.

However, this technique looks simple on paper, but it is in fact extremely heavy during the process. It should be pointed out that selecting a new sample implies computations in order to make the data workable. To remedy this situation, a data processing must be introduced. Data processing takes raw data to prepare it for another processing procedure. It transforms the data in a workable format.

3.3.2.2 Underfitting

The opposite problem that can occur is the underfitting. An underfit model ignores the practice from the training data and does not succeed in generalising to new data. Obviously, an underfit machine learning is not an adequate model and does not have a good performance.

This case can occur when the model is too simple. For instance, there is too few data or the regularisation does not work. On figure 3.5, an example of an overfit model can be seen on the left. In this case, the model does not take into account samples and therefore, a straight line is drawn.

In the case of this experiment, an underfit model is not witnessed. In fact, there are enough samples on the plot and most of them do not move apart from each other. To put it simply, for each iteration, the error is not that big between data.

Chapter 4

Analysis and Results

4.1 Vulnerable dataset analysis

To begin with, the first analysis starts with the tokenization of vulnerable files from the initial vulnerable dataset. In fact, it is interesting to point out some type of tokens found in vulnerabilities. As a reminder, before implementing new tokens, the basic tool C-tokenizer was only recognising 19 different types. Table 4.1 records the number of tokens per type found in the 1526 vulnerabilities inside 922 files. The column "Filtered" shows statistics without taking into account useless tokens like white spaces and comments (these tokens are considered as noise).

Contrary to "square" token number, it is interesting to note that the "open curly" token number is bigger than the "close curly" token number on figure 4.1. It shows that a vulnerability has more probability to be at the beginning of a function or a statement declaration. This latter can be a loop or an "if statement" for example. And logically, a vulnerability has less chance to be found at the statement's end.

Another interesting point concerns comments which leads to multiple interpretations. According to figure 4.1, there are about 0.3% of comments. On the one hand, it does not make sense to see a comment part in a vulnerable dataset knowing that this part of the code will not engage an error. But on the other hand, it shows another aspect of the source code. In fact, it leads to think that the comment section next to the vulnerable part is either missing or its content does not change. The first hypothesis can be easily verified because a lot of code is not commented. The second hypothesis is not reassuring because it means that the commented logic of the vulnerable part stays inside the vulnerable file. To verify it, an in-depth analysis of the code is required.

Token type	Num. of tokens	Percentage	Filtered
Identifier	14387	33.47%	44.26%
Operator	10594	24.65%	32.59%
White space	10351	24.08%	-
Close parenthesis	2686	6.25%	8.26%
Open parenthesis	2669	6.21%	8.21%
Number	869	2.02%	2.67%
Open curly	336	0.78%	1.03%
Close square	244	0.57%	0.75%
Open square	244	0.57%	0.75%
Close curly	213	0.49%	0.65%
Quote	172	0.40%	0.53%
Area comment	126	0.29%	-
Directive	61	0.14%	0.19%
Line continue	21	0.05%	0.06%
Char	7	0.02%	0.02%
Area comment continue	4	0.01%	-
Char continue	1	<0.01%	<0.01%
Line comment	1	<0.01%	-
Word	0	0%	0%
Total	42,986	100% (42,986)	100% (32,504)

Table 4.1: Initial vulnerable dataset statistics

The noise represents 24.38% of the unfiltered vulnerable dataset. Moreover, identifiers and operators are the most common types of token because they represent 58.12% of the source code. For that reason, some of these tokens were specialised as described in section 3.2.2.1.

4.1.1 Operator and identifier specialisation

As mentioned before, operator and identifier tokens are very common inside vulnerabilities. Thus, new tokens were brought because creating more specialised ones would give more diversity.

On table 4.2, there are two sub-tables. On the left side, it is related to operators and the right side is about identifiers. Not all operators and identifiers are shown because the table would be huge. This is why the two sub-tables only displays the ten most used tokens. In addition, a usage percentage is displayed for each token. For example, on table 4.2, the operator token "==" is considered as relational type. Plus, among all the operator tokens, the token "==" represents 2% of usage. The last line of each sub-table indicates the percentage of other tokens.

Token	Operator type	Pct
->	Member and pointer	29.41%
=	Assignment	24.25%
.	Member and pointer	12.24%
-	Arithmetic	5.21%
!	Logical	4.43%
+	Arithmetic	4.34%
|	Bitwise	4%
>	Relational	2.46%
<	Relational	2.43%
==	Relational	2%
Other	-	7.23%

Token	Identifier type	Pct
if	Flow control selection	22.89%
struct	Structure	13.2%
int	Data type	12.53%
long	Qualifier	10.79%
unsigned	Qualifier	9.17%
return	Return type	8.66%
sizeof	Data size	4.07%
goto	Redirecting flow	3.76%
void	Return type	3.76%
static	Storage class	3.28%
Other	-	4.61%

Table 4.2: Operator and identifier types : top 10

Some findings emerge following the analysis of the most encountered tokens in the vulnerable dataset. Firstly, the flow control selection is found one in three times in the total of identifier table. Typically, the "if statement" and the relational operator type are frequently modified. This fact emphasises a problem with conditions. Put in another way, vulnerabilities involves a big part in conditional statements.

Accordingly to table 4.2, a vulnerable part of software will probably contain a source code linked to structures. The fact that on top of the two sub-tables there are three tokens related to structure ("->", ".", "struct") leads to saying that C structure will be often adjusted when a vulnerability is discovered. In other words, this acknowledgement points out that the program structure is usually the subject to error.

Regarding to genetic programming, the table 4.2 put forward an important point for the future tests. In fact, some genetic programming techniques replace a specific token with another. Obviously, it is interesting to change some operators between them. For example, swapping arithmetic operators between them is a basic mutation technique.

However, when it comes to identifier types, that is another question. Replacing a specific identifier token is more difficult in some ways. In fact, in many cases these identifiers are very specific to the environment. For instance, the table 4.2 shows that the most used identifier tokens are closely related to the context. Consequently, some genetic techniques are counterproductive for the identifier token.

Hence, the specialisation of the identifier token is useless but the operator one is interesting to explore. Furthermore, the precision is theoretically valu-

able but it must be used with caution in this case. Too much accuracy will significantly decrease the number of results because the particular and precise context of one vulnerability is sometimes too uncommon. The generalisation of certain tokens can be more advantageous than it appears.

4.1.2 Vulnerable pattern size

Before altering any vulnerable pattern, it is interesting to inspect pattern sizes inside the initial vulnerable dataset. Considering the fact that a pattern is a flow of tokens, it is useful to know some characteristics of the vulnerabilities like the size. This latter is defined by the number of tokens inside a pattern as explained in section 3.2.4.

The figure 4.1 shows the number of patterns without noise per size (as a reminder, the noise represents with spaces and comments). To be more precise, when it comes to the vulnerable pattern size from the vulnerable dataset, there are:

- **Mean**: 21 tokens per pattern;
- **Median**: 9 tokens per pattern.

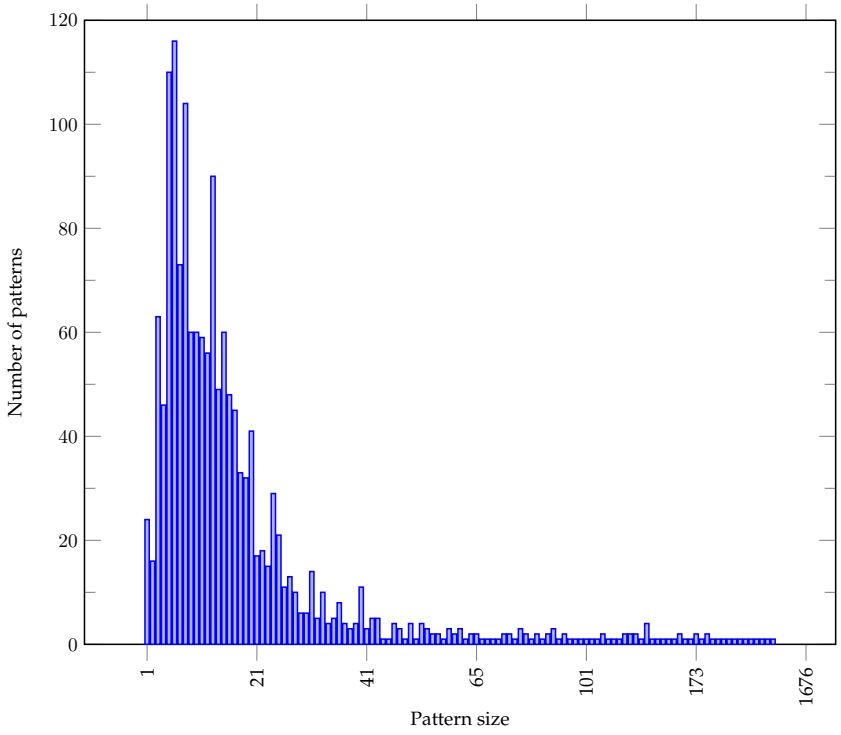

Figure 4.1: Number of patterns per size

This is a fundamental information for the continuation. First, taking a small pattern would provide a great result but it would limit possibilities. In fact, a small pattern would always give trivial outcomes with a genetic programming because mutant and offspring variation would be minimal compared to parents.

Secondly, taking a big pattern would involve a poor result. Actually, there is little likelihood to find a considerable result due the fact that a bigger pattern is just harder to discover. Plus, with a big pattern, the genetic programming would produce a huge population because there are a lot of possible combinations.

As a consequence, all experiments are set up with a pattern size between two selected values:

$$\boxed{30 \leq size(pattern) \leq 80}$$

Outside these two values, there is no point to run tests or scan results. Also, it is interesting to note that the process of searching a pattern in files is obviously faster with a small size pattern. A big one requires a lot of computing power and it takes while to have results.

Another realisation about pattern sizes concerns the fitness score. Experiments showed that a small size pattern gives good fitness scores and a big one delivers bad fitness scores. This is also explained by the fact that small size patterns can be found in many files unlike the bigger ones. Thus, all the difficulty lies in pattern size choice.

4.2 The technical environment

Before introducing the core of the experiment, this section explains the technical side with another perspective. To put it simply, this section presents major technical components and how the experiment is made.

To begin with, the main program which handles the vulnerable dataset content is written Java 8. This program will extract the useful content and put it inside an internal structure. Then, it will tokenize the vulnerable data and store it.

From this point, the Java application passes the baton to the Python application. This latter is an upper layer which controls the Java solution with Py4J[1] - a bridge between Python and Java. With Py4J, the Python program can dynamically access Java objects in a Java Virtual Machine.

The main reason of the Py4J presence is the fact that DEAP framework is written in Python. In summary, the Python program takes via Py4J vulnerable tokenized data from the Java structure in order to use them with DEAP framework. The result is generated with the Python program.

[1] https://www.py4j.org/

The above approach is chosen because the subject of this work was amended during its development. In other words, the DEAP framework solution arrived in the process at the end of the Java solution development. In order to save development time, Py4J was the best compromise.

Another interesting technical point to note concerns the fitness calculation. For each fitness process, there is a calculation involving a multiple search in data. In the normal case, this computation is sequential and is time consuming. In order to enhance this process, the parallel computation technique is used in Python. This module permits to create processes and takes full advantage of multiple processors on a computer. Plus, this technique is complementary with the DEAP multiprocessing module. Hence, the algorithm is improved and takes less time to generate results.

4.3 The experiment

4.3.1 Data pre-processing

As presented in section 3.2.2.1, the tokenization tool provides set of tokens on output by applying a data transformation for every string inside a source code. Unsurprisingly, this process is time consuming for most of the files. And to add to that, there is a read/write process for each file. This latter is a mandatory step which does not accelerate the whole operation.

Thus, some concepts and techniques from data mining must be applied [35]. A data cleaning routines are set up in order to "clean" the data because the dirty information can cause confusion for the mining procedure, emerging in uncertain output. Plus, a data reduction provides smaller but workable data.

One of the most time consuming procedures concerns the random selection of files inside the Linux kernel and its tokenization. In fact, each time a new population of patterns is created, a new set of random files is selected in order to prevent the overfitting case. Here it is in detail the procedure:

1. Catalogue files in the Linux kernel;
2. Select 30 random files from the catalogue;
3. Launch the node command for the C-tokenizer;
4. The C-tokenizer writes a new file containing a set of tokens with noise;
5. A file read process is performed for the C-tokenizer output;
6. The data are cleaned.

It is obvious that the whole procedure for each new population is heavy in every sense of the word. Therefore, the pre-processing consists in performing this whole procedure hundreds of times before the experiment in order to generate "prepared" data. In other words, there is a pre-processing action that creates a large number of files containing a set of cleaned data. Instead of executing all these points listed before, the program has to load files containing

sets of tokens. And of course, the prepared data is always different for each new population.

The same logic is applied with vulnerabilities and vulnerable files from the vulnerable dataset. The fact that it is the starting point of the experiment and that its content is always the same, the pre-processing can accelerate the whole experiment by generating prepared data.

To summarise, here is all the different kinds of prepared data, with the aim of speeding the experiment. In all cases, it is a stored file with a list of tokens without any noise:

- A set of random files from the Linux kernel;
- Chunks that represent vulnerabilities from the vulnerable dataset;
- Files labelled as vulnerable from the vulnerable dataset.

4.3.2 Main process

The starting point for the core process is the listed known vulnerabilities. The first step is ranking all of them with a fitness score. Therefore, each vulnerability is associated with a fitness score and a ranking can be made. This classification is decisive because patterns with best scores are part of the selection process.

Once the superior patterns are selected, mutation and crossover operators are applied. These two processes generate a whole new population of patterns. Unfortunately, there are a lot of flawed individuals inside this new population. Hence, a data cleaning is compulsory to eliminate them. Then again, the fitness is calculated on the new population and the process begins the cycle again. To summarise, listing 4.1 shows the whole process in pseudo-code.

```
1 initialise population with a set of vulnerable patterns
2 repeat
3         data cleaning
4         fitness evaluation on the population
5         selecting best vulnerable patterns
6         crossover
7         mutation
8 until terminal condition
```

Listing 4.1: Pseudocode of the main process

It is important to note that a data cleaning is performed each time a new population appears. Plus, the terminal condition of the loop is the number of iterations that is chosen at the beginning of the experiment.

Another important note concerns the loop: each new generation is produced asynchronously to a certain extent. As explained in section 2.3.4, an asynchronous generation means that the old and the new generation overlap. But in this experiment, the asynchronous concept is done to a certain degree: only best patterns from the old generation are preserved. In contrast, the syn-

chronous generation does not produce any meaningful results.

At the end of the experiment, the last population should theoretically provide a set of strong vulnerable patterns.

4.4 Vulnerable pattern selection

Each time a new generation is created, a pattern is selected for the mutation process and two patterns are chosen for the crossover procedure. In both cases, it is a random choice based on probabilities. To this end, The Poisson Distribution is used.

Before taking into account the probability regarding the selection process, an observation was made during experiments: taking always best individuals in order to generate a new population leads to nowhere. Instead of creating a personal selection probability, a basic discrete probability distribution is used. It already provides fixed values and can be applied to the selection algorithm in order to boost diversity within the population.

To illustrate a use of this distribution in the context of this work, an example is shown in this section. On figure 4.3[2], the horizontal axis is the index k (the number of occurrences). The vertical axis represents the probability of k occurrences depending on λ (the expected number of occurrences).

When a new generation is created, a fitness process is performed in order to classify from best to worst patterns. In other words, a classification is made where the first element is supposed to be the best one. To produce a new generation, a selection must be performed among the best elements. On figure 4.3, the first 10 elements of this ranked list are selected (k = 10). The figure shows 3 different set of patterns where each element has a given probability to be selected for the mutation/crossover process.

A different view of the Poisson distribution can be seen on table 4.3. For example, the second set of patterns (λ = 3) shows the probability of each patterns to be chosen. For instance, the fourth pattern (k = 4) has 17% of chance to be picked up for the mutation process.

With this technique, patterns are selected depending on their associated probability. Plus, the fact that the expected number of occurrences (λ) is selected randomly each time, the best patterns are not always the ones with the highest probability values (this can be seen on when the λ increases). For example, when λ = 5, the pattern at the first position has only 1% of chance to be chosen by the process. This last point is set up in order to make a new generation wider and more varied.

[2]Source: http://sasnrd.com/poisson-distribution/

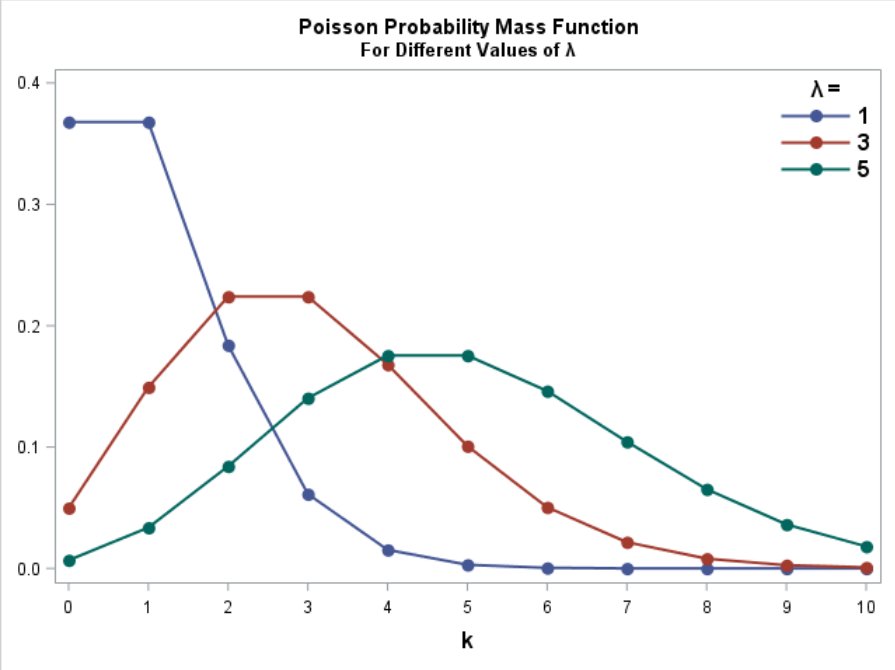

Figure 4.2: The Poisson distribution

		Pattern number (k)										
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
λ	**1**	37%	37%	18%	6%	2%	0%	0%	0%	0%	0%	0%
	3	5%	15%	22%	22%	17%	10%	5%	2%	1%	0%	0%
	5	1%	3%	8%	14%	18%	18%	15%	10%	7%	4%	3%

Table 4.3: Pattern probability selection

4.5 Mutation and crossover used techniques

After selecting unique patterns with a given probability, new individuals must be produced. The new generation will be composed of two kinds of individuals: a set of vulnerable patterns produced with mutation techniques and a set of vulnerable patterns created with crossover techniques. This section introduces used techniques and the data validity on output.

There are all kinds of different techniques when it comes to the mutation or the crossover techniques. Some of them have been presented in section 2.3.5 and 2.3.6. Therefore, there is no need to reinvent the wheel because many methods have already demonstrated their performance.

As explained before, the main evolutionary computation framework used here is DEAP (see section 2.3.7). Concerning mutation and crossover operators, they offer a set of methods in its evolutionary tool. To put it simply, it contains the operators for evolutionary algorithms[3]. Because of this module, it is possible to modify, select and move individuals in their context.

4.5.1 Data validity

As a reminder from section 3.2.2, a vulnerable pattern is represented as a sequence of tokens. And logically, mutants and offspring are represented the same way. However, each new individual generated is not valid regarding the architecture: the suitability is defined by the validity of the token set.

Each recognised token has a unique token number. But a mutation or a crossover process can produce an unidentified token inside the new individuals. Therefore, a data cleaning must be performed on the new generation in order to remove invalid individuals. In short, the law of the strongest is applied: the weakest and most helpless are eliminated.

4.5.2 Mutation operators application

A series of mutation operators are used from DEAP in the context of this work. This section presents different techniques from this framework and their actual usefulness in the context of tokenization.

Shuffles indexes : As its name suggests, it shuffles different tokens inside a sequence. There is no new token produced and therefore, most of new individuals are valid after the data validity verification. But this process has its limit: changing randomly the token order will not make a vulnerable pattern stronger. In fact, there is a lot of chance to have an absence of result.

Uniform integers : This kind of mutation will replace a random token inside a token set by a value uniformly drawn between low and up inclusively. This method can be interesting because it does not really produce a random individual. With a given probability of modification for each token and a given interval of a minimum/maximum token value, the new individual receives a small adjustment.

Polynomial Bounded : This mutation is an implementation of the NSGA-II algorithm [36]. Clearly for this context, the polynomial bounded operator is present for experimental purposes. Therefore, this technique is not used a lot.

[3] http://deap.readthedocs.io/en/master/api/tools.html

4.5.3 Crossover operators application

Overall, for each crossover operators, there are a lot of results. Crossover methods presented in this section are simple to understand and easy to use. In addition, these methods provide diversity for the new generation. But a particular attention must be paid here because sometimes, offspring are totally different compared to parents (especially with the messy one-point and uniform technique). It should not be forgotten that each new generation is supposed to have things in common with the previous one.

One-point crossover : A random position is chosen along the two parents. Then, the parents are cut at the given position, and their end parts are swapped to create two different offspring. On figure 4.3, an example can be seen [37].

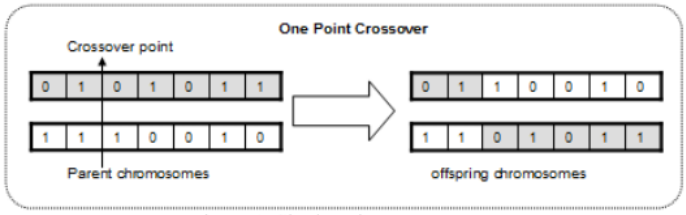

Figure 4.3: One-point crossover

Two-points crossover : The two-points crossover operator choose two points in each parent. Then, the content between these points are exchanged between two mated parents. Figure 4.4 shows an example [37].

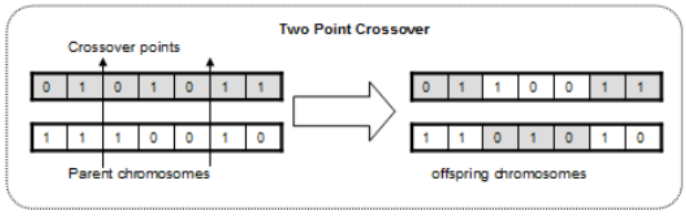

Figure 4.4: Two-points crossover

Uniform crossover : This operator picks values from two parents randomly to create the offspring [38]. An example on figure 4.5 is given where two offspring are composed by uniform crossover.

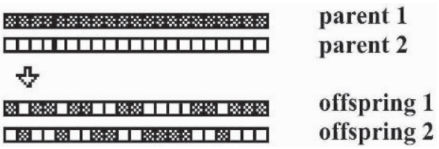

Figure 4.5: Uniform crossover

Messy one-point : The messy one-point crossover use cut and slice operators allowing an offspring to inherit random values from each parent. Surely, this allows new individuals to have duplicate values or missing data [39]. In most situations the individual size on output changes. This kind of mutation produces a lot of valid result but the fact that the size varies every time could potentially raise the question with the vulnerable pattern size as explained in section 4.1.2.

4.6 Results

After having presented details of the main process, this section presents results. As a reminder, the question is : "Can a vulnerable pattern be improved with genetic programming techniques?".

4.6.1 Variable parameters

During the experiment, there are certain number of parameters on background that define the context of the experiment. In other words, technical specifications chosen before starting the experiment are important. Therefore, this section presents different aspects influencing the final result.

4.6.1.1 First vulnerable pattern

First of all, the whole experiment depends on the vulnerable dataset introduced in section 3.1. The first vulnerable pattern is selected will influence the result directly. Without any doubt, there are some vulnerable patterns from the vulnerable dataset with zero impact. In other words, if one of these bad patterns is selected at the beginning, there is some chance that the outcome will not be satisfying. Unfortunately, is it impossible to know in advance if a pattern is good or not.

However, if we want to be more precise, a vulnerable pattern cannot be "bad" or "good". These terms are used here in order to simplify the explanation. In fact, a vulnerable pattern can be potentially too specific: it is attached to a certain context in a particular way. If that occurs, the vulnerable pattern

is too complex and genetic programming techniques do not help. This kind of problem will be compared to the Wild-Caught Mutants issue in section 5.2.1.

4.6.1.2 Fitness process

The fitness function introduced in section 3.3.1 and applied randomly on different files can bring a small problem. It is recalled that the fitness process is based on searching a piece of data inside a file. The problem is the randomness: selecting a random file does not guarantee that the selected search domain fulfil expectations. In other words, if the random file contains too specific data, the fitness process does not provide any result.

This issue can be compared to the first vulnerable pattern selection case. The common factor of randomness could lead to a useless process where the result does not show any interesting data. But in this case, effects are not definitive on the outcome: as explained in section 4.3.1, there is sufficient flexibility because 30 random files are selected. On one side, it gives some room for manoeuvre and on the other side, it prevents the overfitting problem as seen in section 3.3.2.1.

4.6.1.3 Number of jokers

As introduced in section 3.2.2.2, a special token is created and it is called the "Joker". As a reminder, this kind of placeholder accepts any value when the fitness is calculated. Its functioning is simple but there is an important side effect: too many jokers inside a pattern structure leads to biased results.

Logically, at some stage, the algorithm understand that implementing jokers makes a pattern more powerful because it can be exploited more easily during the fitness computation. In this case, the maximum number jokers must be defined before the beginning of the experiment otherwise there are a lot of individuals with just some jokers at different positions. Thus, in the experiment, the number of jokers inside a single pattern cannot exceed 10% of the total pattern size.

4.6.1.4 Number of iterations

The maximum number of iterations is defined before the beginning of the experiment. To put it simply, it is the amount of time that the main process presented in section 4.3.2 is executed. Theoretically, at one point, a population score does not evolve anymore - this means that the algorithm reached its edge.

The algorithm stops if one of the following conditions is met:

1. The maximum number of iterations is reached;
2. The number of failed attempts for one generation is reached.

The second condition is also fixed before running the algorithm. It exists because sometimes, the main process does not produce enough valid new individuals for some reason. Therefore, the algorithm gives a "second try" to create a new population. If it fails after a given amount of time, the algorithm stops. This approach tries to handle a certain degree of randomness.

Regarding the maximum number of iterations, it is more challenging to define one. Once again, it depends on the vulnerable pattern potential: some of them reaches their maximum very fast and other needs more iterations. In a case like this, the algorithm should stop by itself because there is no more gain relating to the fitness score.

4.6.1.5 Top individuals scope

Another parameter to take into account before starting the experiment concerns the scope of the ranked vulnerable pattern list. The algorithm selects three vulnerable patterns using a technique described in section 4.4. Obviously, choosing a vulnerable pattern with a bad fitness score does not lead to valuable results. Therefore, choosing a pattern from a set that contains only vulnerable pattern with an excellent fitness score is more valuable.

This process raises questions on the size of the selected vulnerable patterns. It is true that the vulnerable patterns with the highest fitness score have more potential. But, selecting always the top of the ranked list does not bring diversity in the new population. Hence, sometimes it is necessary to choose a vulnerable pattern that is not on the podium regarding its fitness score.

By default, the experiment only takes into account the first ten vulnerable patterns from the ranked list. In that way, the new population has the insurance in terms of diversification. However, this given number can become a problem if there is not enough new individuals. This situation can happen when the size of the selected vulnerable pattern is too big. This latter becomes too specific for the experiment and does not produce enough valid new individuals. The number of failed attempts for one generation presented in section 4.6.1.4 limits the damage.

Plus, as explained in section 4.3.2, the generation from the main process is asynchronous. By default, the experiment takes into account just 2 vulnerable patterns. In other words, there are 2 individuals taken from the generation N and put in generation $N + 1$.

4.6.1.6 Number of generated individuals

Previously, section 4.5 presented different kinds of genetic programming techniques. Technically speaking, the number of generated individuals cannot be predicted precisely. As a result, this section presents a set of variables that could affect directly this number regardless the used technique.

Each technique can be assigned with a given number of repetitions. This latter decides how many times a technique will be called for one selected vulnerable pattern. Also, it should be noted that the size of the selected vulnerable pattern affects the outcome. In other words, genetic programming techniques generate fewer new individuals if the selected vulnerable pattern is small and these same techniques create a lot of individuals with bigger ones. Here, the number of repetitions is between $\sim$500 and $\sim$1000 depending on techniques.

Besides that, genetic programming presents another important variable: the probability of each token to be changed. Considering the fact that a vulnerable pattern is composed of tokens, an important decision must me made. In fact, a high probability means that the new individual is almost different and in the same way, a small probability value implies that the new individual would be a simple copy. In the experiment, this value is between 5% and 10% for each token depending on techniques.

4.6.1.7 Fitness mean evolution

To get a better idea of the evolution during the experiment, a simple indicator is needed. For that purpose, a fitness mean is calculated for each population in order to have a quick look on the population "strength". It gives an understandable visualisation for the next section when it comes to the graph study.

Given that the randomness is strong during the experiment because of all the parameters listed below, the fitness mean value fluctuates a lot sometimes. Undeniably, if the fitness value increases during the experiment, the outcome is fantastic. Unfortunately, this is not always the case and the fitness mean value oscillates very often in both ways. Worse still, a fitness mean value can drop significantly from one generation to another because the main process selected patterns with no potential and produced a bad set of individuals.

This case scenario happens from time to time and can lead to bad results. To remedy this situation, a simple technique is implemented and it is based on the comparison between two fitness mean values. To be more specific, a juxtaposition is made between a fitness mean value from generation N and $N + 1$: if the fitness mean value drops too much, the main process starts again the procedure. Obviously, the process does not repeat indefinitely the generation procedure. Hence, if the fitness mean score loses 100% of its value, the operation will restart up to ten times.

4.6.2 Visualisation

With all the parameters chosen, the experiment produces a result in text format. The content has multiple lines with three indications concerning a vulnerable pattern:

1. The generation number;
2. The fitness score;
3. The token sequence.

Clearly, in order to generate a graph, the first two points are needed. Therefore, the process reunites all patterns for one given generation and calculates the fitness score mean. The purpose is simple: see if the mean improves over time.

But displaying only the mean on the graph is not enough to get an idea of the general outcome. Therefore, the second step concerns the concept of curve fitting with a model. It is the process of creating a curve that has the best fit to a series of data points. Here, the fitness score means are the series of data points.

The presented models have polynomial features of distinct degrees. For instance, a polynomial with degree 1 is not useful because a linear function is not adequate to fit the training samples. In case of this experiment, a polynomial of degree 4 is used. The degree value is determined according to the mean squared error (MSE) : a high value means that the model does not generalise accurately from the training data. Thus, the degree value is chosen when the MSE value is at its minimum.

As explained previously on section 4.6.1, there is a large number of parameters to take into account. Thus, different kinds of parameters combination is tested during experimentation and the result can be examined from multiple angles. The most important parameters are as follows:

- The maximum number of generations that the experiment creates;
- The minimum size of the selected patterns;
- The number of top patterns considered for GP techniques;
- The mean squared error (MSE).

The outcome of the global algorithm is shown on figure 4.6 where the fitness score mean for a given generation is a green dot and the model is the blue curve. This figure as itself shows four results with each time a different set of parameters as presented on table 4.4. The number of generations goes from 100 to 1000 in order to demonstrate that the algorithm is interesting independently to the number of iterations.

	Figure 4.6a	Figure 4.6b	Figure 4.6c	Figure 4.6d
Nb of generations	100	200	500	1000
Minimum size	30	40	40	35
Nb of top patterns	8	8	8	8
MSE $\pm$ Error	0.0351 $\pm$0.0033	0.0241 $\pm$0.0138	0.0298 $\pm$0.0020	0.0436 $\pm$0.0030

Table 4.4: Different parameters depending on the figure

At the first look, figure 4.6 indicates positive results for the most part. In fact, models presented on this figure have a promising outcome. Clearly for each model, the fitness score mean of the last population is greater than the first one. Hence, it proves that genetic programming techniques improves the fitness score mean.

Interesting observations can be made on figure 4.6 (a), (b) and (c) because they have a certain likeness. At the beginning, there is an expansion: the algorithm selects vulnerable patterns with the best fitness score and the next few generations displays directly a better fitness score mean. Then, there is a contraction presenting a trough: the fitness score mean seems to decrease. But the recovery comes after and the algorithm starts to produce strong vulnerable pattern as it can be seen distinctly on figure 4.6 (b). Obviously, the algorithm learns with the training data and produces only "powerful" vulnerable patterns.

On the opposite, figure 4.6 (c) shows a logarithmic evolution with no contraction at all on the model. On the one hand, it is a good result because the fitness score mean does not decrease but, on the other hand, the model does not have the potential to evolve more.

Additionally, the minimum pattern size is an important part of the experiment because it defines a portion of the vulnerable dataset that is ignored. As analysed in section 4.1.2, most of the vulnerable patterns are composed of fewer than $\sim$20 tokens (these patterns do not take into account white spaces or comment sections). But, as presented in table 4.4, the minimum is 30 and the maximum is 40 tokens. It is a conscious choice because the algorithm is designed to find new vulnerable patterns from the well-known. Furthermore, during experiments, observations were made about the number of tokens in a pattern:

1. Obviously, small vulnerable patterns produce less "new" patterns with genetic programming. This means that there is less probability to have new vulnerabilities and observations were made about the number of tokens in a pattern: as a result, new individuals will progressively resemble to the first generations.

2. Small vulnerable patterns have more chance to be discovered in non-vulnerable files because they are too generic. For instance, many vulnerabilities relate to one changed line where a function name is modified. This kind of case is not representative inside a tokenization process and therefore, it is useless to analyse it.

3. Technically speaking, a pattern with a large number of tokens is more time consuming during the experiment. This case can become a huge problem on many levels:

    - A gigantic populations is created and needs to be filtered.
    - The fitness process where a pattern is searched inside files takes a fairly long time to calculate a value.

Observations listed above brings a simple conclusion to write but hard to implement in the algorithm: "A pattern has to be big enough but not obese". Clearly, when it comes to talking about the number of tokens in a pattern, the real difficulty lies to select the right interval.

Another important aspect about the experiments that needs to be explained is about the showed randomness. As explained before, there are a lot of configured parameters that concerns probability in general. For example, there is a value that determines if a given token is going to be mutated or if a vulnerable pattern is selected. Consequently, like with the theory of evolution, there are some aspects that cannot be calculated in advance.

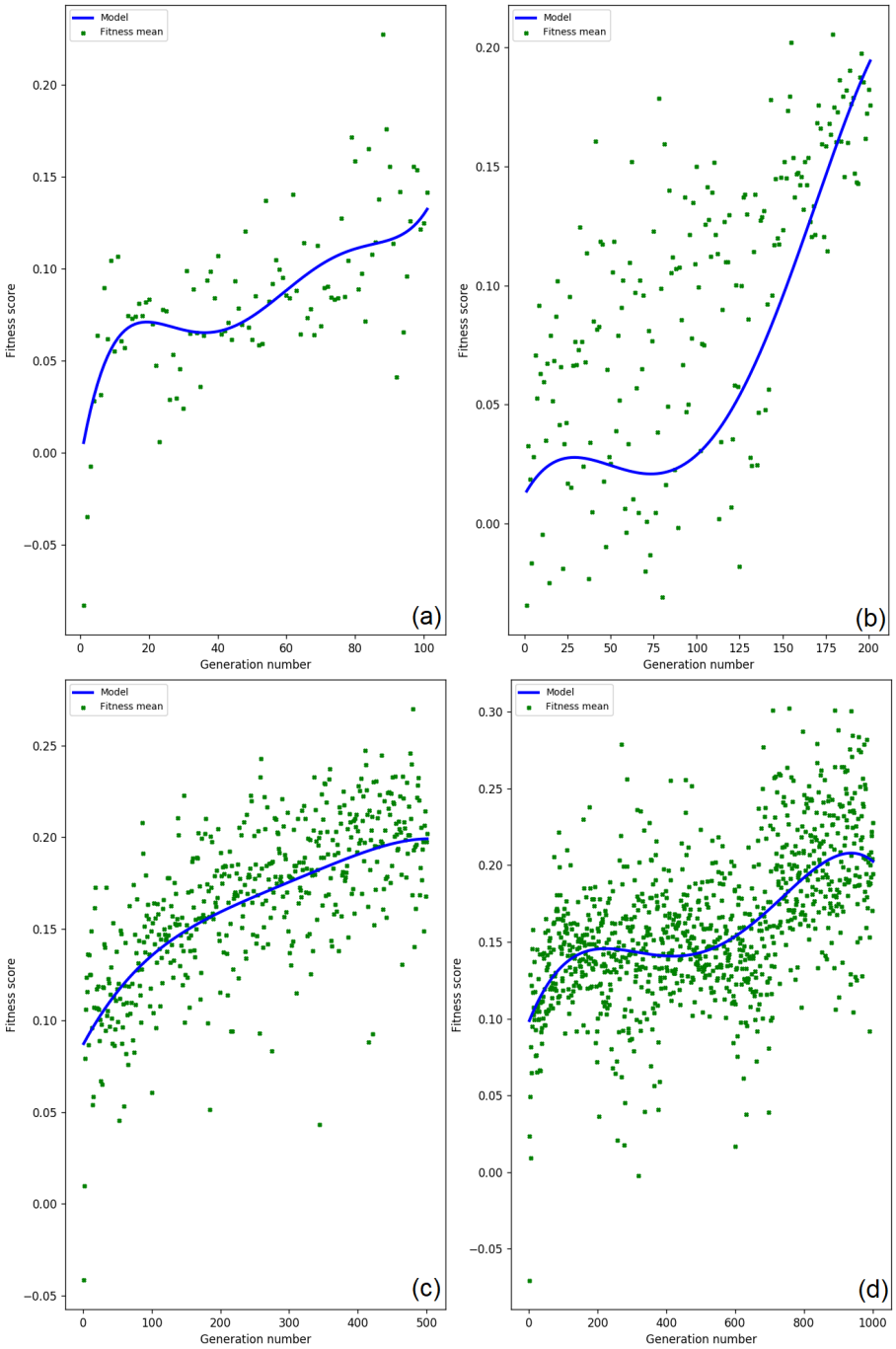

Figure 4.6: Results with different numbers of generations

4.6.3 Coverage investigation

Once experiments are done, there is another facet to explore. Figure 4.6 shows in each result that there are some vulnerable patterns with a high fitness score. This section is going to investigate more closely these individuals by inspecting global coverage independently to the generation number.

At the end of the experiment, it is interesting to inspect the new individuals with the vulnerable dataset. As explained in section 3.1, there is a list of recognised vulnerable file. This list is the central point for the detailed investigation in order to see the overall coverage of mutants and offspring. In other words, it is interesting to see if some good individuals cover a lot of files or not.

Therefore, this kind of analysis shows if patterns with good fitness scores perform well on the vulnerable dataset and if they are connected somehow. The concept of connection relies on the vulnerable file: two patterns are connected if they are found in the same set of files.

The experiment on this section directly concerns results presented in table 4.4, section 4.6.2. Each experiment in this section selects $\pm 5\%$ of the total population and keeps only individuals with the best fitness. For each pattern from the results, the fitness score is calculated again and concerned vulnerable files are stored. Finally, patterns are compared in order to found equivalency.

Data in tables 4.5, 4.6, 4.7 and 4.8 are classified by the number of concerned files. They have these following information:

Pattern ID Patterns are no longer treated according to their respective generation. Therefore, an ID is associated with each pattern. A pattern with an ID_1 means that it is the first of the list in terms of coverage. Tables have missing ID at some point and it indicates that the given pattern affects the exactly same set of vulnerable files. For example, in table 4.5, there is no pattern with an ID_2 because it refers to the exact same list of vulnerable files as ID_1.

Fitness score It indicates the fitness score value (see section 3.3 for the formula). The evaluation strategy is similar to the main process as seen in section 4.3.2. In this experiment, the fitness evaluation is performed only on good patterns.

Number of concerned files When a vulnerable pattern is found inside a vulnerable file, it is considered as a concerned file. A high value indicates that a given pattern is found in a lot of vulnerable files.

Percentage of concerned files The number of vulnerable files from the vulnerable dataset is fixed : there are 922 vulnerable files listed. Therefore, the percentage represents the coverage of a given pattern.

Number of same-scoped patterns As explained before for the "Pattern ID", some of the patterns concerns the exactly same set of vulnerable files. Instead of listing them with details, this column indicates the exact number of same patterns and it provides a link between different rows of "Pattern ID". For instance, in table 4.5 for ID_1 the value for this column shows 10 same patterns: in other words, patterns ID from 2 to 11 covers exactly the same set of vulnerable files.

Pattern size It shows the size of the given pattern. As a reminder, the whole experiment only considers useful tokens: white spaces and comment sections are filtered.

Precision It is the fraction (or percentage) of retrieved documents that are relevant. Based on the confusion matrix presented in section 3.3.1:

$$Precision = \frac{TP}{TP + FP}$$

Recall It is the fraction (or percentage) of the relevant documents that are successfully retrieved. Based on the confusion matrix:

$$Recall = \frac{TP}{TP + FN}$$

Pattern ID	Fitness score	No. of concerned files	% of concerned files	No. of same scoped patterns	Pattern size	Precision	Recall
1	0.423	276	30%	10	30	53%	30%
12	0.283	232	25%	0	31	56%	19%
13	0.388	198	21%	7	34	56%	19%
21	0.327	179	19%	1	36	67%	27%
23	0.302	178	19%	0	39	78%	24%
24	0.42	174	19%	4	39	47%	15%
29	0.327	143	16%	0	43	58%	13%
30	0.363	139	15%	0	43	43%	10%
31	0.42	138	15%	3	44	73%	11%
35	0.392	133	14%	18	44	54%	15%
54	0.327	119	13%	0	47	50%	14%
55	0.363	113	12%	6	49	54%	13%
62	0.267	98	11%	0	53	75%	10%
63	0.302	96	10%	0	54	80%	14%
64	0.363	95	10%	16	55	54%	13%
81	0.333	85	9%	7	59	51%	10%

Table 4.5: File coverage for 100 iterations

Pattern ID	Fitness score	No. of concerned files	% of concerned files	No. of same scoped patterns	Pattern size	Precision	Recall
1	0.233	176	19%	1	40	59%	17%
3	0.250	145	16%	0	40	75%	10%
4	0.392	143	16%	115	42	51%	16%
120	0.327	128	14%	0	45	80%	13%
121	0.327	123	13%	0	46	80%	13%
122	0.333	119	13%	26	48	55%	12%
149	0.302	99	11%	0	50	60%	14%
150	0.333	98	11%	19	54	54%	12%
170	0.302	89	10%	0	55	86%	20%
171	0.333	85	9%	38	58	52%	13%
210	0.333	79	9%	133	63	52%	10%
344	0.363	68	7%	129	69	53%	7%
474	0.302	62	7%	15	74	52%	7%
490	0.302	60	7%	15	78	84%	7%

Table 4.6: File coverage for 200 iterations

Pattern ID	Fitness score	No. of concerned files	% of concerned files	No. of same scoped patterns	Pattern size	Precision	Recall
1	0.333	176	19%	10	40	51%	17%
12	0.392	174	19%	24	40	49%	17%
37	0.327	153	19%	0	42	60%	20%
38	0.392	152	17%	2	42	55%	22%
41	0.268	145	16%	0	41	63%	17%
42	0.42	143	16%	16	41	51%	17%
59	0.392	140	15%	3	41	56%	13%
63	0.392	139	15%	3	43	57%	17%
67	0.333	138	15%	38	45	52%	14%
105	0.447	133	14%	46	45	53%	14%
152	0.392	122	13%	0	46	63%	17%
153	0.327	120	13%	1	47	56%	13%
155	0.363	119	13%	7	48	58%	10%
163	0.327	116	13%	6	48	53%	10%
170	0.363	115	12%	38	50	54%	14%
210	0.327	113	12%	25	50	55%	13%
236	0.363	98	11%	6	53	56%	10%
243	0.363	96	10%	15	55	58%	13%
259	0.333	95	10%	6	54	49%	10%
266	0.363	85	9%	27	60	58%	12%

Table 4.7: File coverage for 500 iterations

Pattern ID	Fitness score	No. of concerned files	% of concerned files	No. of same scoped patterns	Pattern size	Precision	Recall
1	0.388	200	22%	0	35	69%	37%
2	0.417	198	21%	0	36	40%	20%
3	0.447	186	20%	0	35	39%	17%
4	0.5	179	19%	95	37	50%	17%
100	0.447	178	19%	9	39	52%	18%
110	0.392	176	19%	4	37	50%	17%
115	0.363	161	17%	0	37	50%	10%
116	0.333	145	16%	2	40	66%	18%
119	0.447	143	16%	19	44	56%	17%
139	0.388	138	15%	2	43	59%	13%
142	0.333	123	13%	0	46	67%	13%
143	0.42	119	13%	14	48	53%	14%
158	0.392	116	13%	1	48	63%	12%
160	0.333	115	12%	0	50	80%	13%
161	0.392	98	11%	17	51	59%	8%
179	0.363	96	10%	1	53	58%	7%
181	0.363	85	9%	4	58	72%	11%
186	0.363	79	9%	6	63	48%	7%
194	0.363	85	9%	4	58	58%	10%

Table 4.8: File coverage for 1000 iterations

First, table 4.5 is not very representative of the global outcome because the number of individuals is not significant. For example, the pattern ID_1 covers over 30% of the vulnerable dataset. Compared to other tables first pattern of the list, this is too high. Clearly, this shows that figure 4.6(a) does not produce enough data to be analysed.

On the other hand, tables 4.6 and 4.7 display the same outcome in a certain manner: both have generic and specific patterns. It is interesting to note that ID_4 from table 4.6 covers the same set of files as 115 other patterns from this generation. This can be attributed to two factors.

1. A side effect of implementing the joker concept. A pattern is considered different to another if its internal structure has a distinct set of tokens. As a result, different patterns may affect the same set of files if they have a certain correspondence through jokers.

2. Apart from the joker case, there is another situation where a pattern possesses the same trait as another. For instance, a pattern with a given size of N tokens can potentially affect the exactly same files as a pattern with a size of N+1 tokens. In other words, a token sequence could probably bring the same result as another if there is not a large gap concerning their size.

Table 4.8 presents an interesting coverage. The first three patterns covers 22%, 21% and 20% respectively of the vulnerable dataset - but the number of same patterns is zero. Moreover, their fitness score are very high. One interpretation would be that these three patterns are generic because the concern a large number of files and they are uncommon because basically there is no other similar patterns.

Being generic and uncommon sounds counter-intuitive as first. On the one hand, this kind of pattern could apply on many vulnerable files but, on the other hand, it is likely to be very specific to the context. Further research is needed on that point.

Overall, coverage results vary depending on the number of iterations and there is no particular correlation between each column. However, it is worth noting a certain connection between the number of concerned files and the pattern size. In fact, the size decreases as the percentage of concerned files increases. Clearly, this can be explained by the fact that a smaller pattern can be more easily found in files. Also, an interesting observation can be made after a closer look on table 4.4: the algorithm seems to increase patterns size over generations. New generations are composed of bigger specialised patterns with a strong fitness score but with less files coverage.

No. of iterations	**Avg. fitness score**	**Avg. precision**	**Avg. recall**
100	0.35	60%	16%
200	0.32	64%	13%
500	0.36	57%	14%
1000	0.39	58%	15%

Table 4.9: Average results

On another note, table 4.9 displays the average fitness score, precision and recall. As a reminder, precision is measuring what fraction of predictions for the positive class is valid. Concerning the recall, it telling how often predictions actually capture the positive class.

The precision value answers the question "Of all things that were identified as vulnerable, how many were actually vulnerable ?" and the recall measure is "Of all the things that are truly vulnerable, how many did it identified?".

Here, around $\sim$60% of predictions for the positive class are valid and $\sim$15% of predictions capture the positive class. Clearly, the precision is not extremely high and the recall is low. In other words, it means that $\sim$60% of the patterns that were labelled as vulnerable were indeed vulnerable. Regarding the recall, only $\sim$15% of the truly vulnerable pattern were caught by the algorithm.

However, the classification judgements can be trusted partially. But it is very conservative: it means that a lot of vulnerable patterns remained unidentified.

Chapter 5

Related work

5.1 Problem statement

This section presents some related work to the subject. To be more specific, targets concern some concepts presented in this work. The initial point is to inspect a perspective of an actual working procedure and then try to apply it in the context of the local vulnerable dataset as presented in section 3.1. Plus, another point to explore is the use of tokens as explained in section 3.2.2. And finally, the last point to consider is the genetic programming as introduced in section 2.3.

5.2 The Care and Feeding of Wild-Caught Mutants

The first section is related to the "The Care and Feeding of Wild-Caught Mutants" published by Brown et al. from the University of Wisconsin [40]. This paper submits a technique to increase performance of the mutation testing. It implements a method for conceiving possible faults that are linked with changes made by actual developers. Hence, this technique allows the tester to have more conviction that the test cases are sensitive to changes that have been studied.

The main objective of this research project involves a new examination of the mutation testing by making use of mutation operators that are more closely resembling defects injected by real developers. To put it simply, the Wild-Caught Mutants solution is a method for creating potential faults that are linked with defects created by actual developers.

5.2.1 Wild-Caught Mutants toolchain

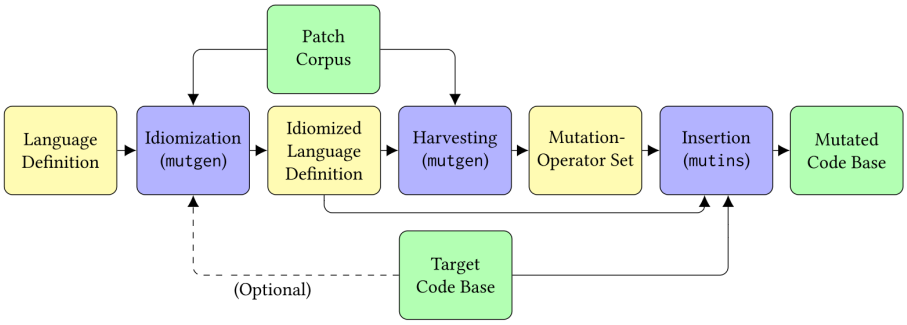

Figure 5.1: The mutgen / mutins toolchain

Language definition: A language is defined with a set of operators (O), keywords (K), quote delimiters (Q) and comments (C) as seen on listing 5.1. Basically, the presented language parser is a lexical analyser and it allows to spot tokens with high accuracy. In other words, a lexical analysis generates a set of tokens identified by the rules in the language definition.

```
1 K auto break case char const continue default do double else enum
2 K extern float for goto if int long register return short signed
3 K sizeof static struct switch typedef union unsigned void
4 K volatile while
5 O = += -= *= /= %= &= |= ^= <<= >>= ++ -- + - * / % ~ & | ^
6 O << >> ! && || == != < > <= >= [ ] -> . ( ) , ? :
7 Q ' "
8 C /* */
9 c //
```

Listing 5.1: Language definition for C

Idiomisation: The language definition covers language keywords and operators. Yet, some identifiers are used too many times and as a consequence, they must be considered as additional keywords. These specific identifiers are considered as "idioms" and the process of locating them is "idiomisation". The process of idiomisation can recognise idiomatic keywords that appear barely in the system under test. As a result, the idiomisation is optional.

Harvesting (mutgen): It is the extraction process of reusable mutation operators. Referred as the "harvesting", it creates a mutation operator set from a corpus of diff-formatted code patches (see section 3.1.3 for a diff example). In practice, mutgen classifies mutation operators by cutting off small transformation from the revision history received in input.

On listing 5.2, an example of the command line is given. The mutgen process takes three parameters into account: the language definition in listing 5.1, a diff revision input (see example in listening 3.3) and an output name.

```
1 ./mutgen -d clanguage.def -i input_diff.git -x output.dat
```

Listing 5.2: Mutgen command

After running the mutgen command, a file on output is generated. A line example of its content can be seen in listening 5.3. At the first approach, it seems incomprehensible but that is not the case. In fact, the line can be divided in two: what is on the left and what is on the right of the "@". The right part concerns an example of the pattern which is on the left side. In each case, the symbol ":" indicates the code mutation. In this example, a pointer is eliminated on the second identifier $2 and it can be observed on the variable called *buf*.

```
1 M $1 .= .*$2 : $1 .= $2 @ p = *buf; : p = buf;
```

Listing 5.3: Mutgen output

Insertion (mutins): This is the process of applying operators to the source code. In other words, this insertion tool applies mutation operators to a source code different from the one used during harvesting. Therefore, on output, the source code will be mutated.

Mutins operates by tokenizing the source code input file using the same language definition as presented in listening 5.1. Then, it chooses a mutation operator from the mutation operator set. It has to be noted that the selection is done randomly but it can be specified by the developer. After, the process insertion tries to match the mutation operator's pattern. If there is a match, mutins alters the tokens in the source file.

In order to illustrate the insertion process, the listing 5.4 demonstrates the mutins command. It takes four arguments: the output from listing 5.2, the mutant index number N_1 to be used, the specific index for insertion N_2 and the result file name.

```
1 ./mutins -x output.dat -m N1 -i N2 -t result.c
```

Listing 5.4: Mutins command

5.2.2 Vulnerable dataset association

Once the main concept of the Wild-Caught Mutants is assimilated, the next step is to find a potential combination with the vulnerable dataset as described in section 3.1. In fact, using an existing project with confirmed results can be very profitable.

As described in the previous section 5.2.1, the input diff revision has similarities with the vulnerable dataset. In fact, the input file in the form of comparison of two different files is the same format. Thus, it is interesting to see if using a vulnerable dataset from Linux kernel can be coupled with the Wild-Caught Mutants solution.

After extracting the vulnerable parts from the vulnerable dataset, the experiment consists to see if the mutgen process could produce an interesting mutant pattern. In other words, the harvesting aspect from the Wild-Caught Mutants is used to see if the output is promising. Plus, the vulnerable dataset includes files written in C and the language definition for C is provided for the tool. It is thus clear that the tool could potentially be complementary.

Regrettably, the Wild-Caught Mutants solution cannot fit with the vulnerable dataset. This conclusion can be explained easily by analysing the output data from the test case and from the vulnerable dataset. To put it simply: at first, the test concerned files provided by the project and secondly, the test concerned files provided by the vulnerable dataset.

In the first instance, the experiment takes into account all the artifacts[1] provided by the project. Over 53 elements are generated from different sources. The content analysis revolved around the biggest files: in this way, many cases of tokenization can be observed instead of few ones from smaller files. The main observation spotlights the fact that many mutants were very basic. In other words, the developer mistakes involved in most cases the lack of attention.

Taking into account the first point brings to the fore the problem of the second case involving the vulnerable dataset: most of vulnerable chunks reported with the CVE-NVD database are complex. In fact, the level of programming is not the same and it is problematic. In other words, vulnerabilities from the Linux kernel are hard to tokenize with the Wild-Caught Mutants tool.

The experiment with the vulnerable dataset brings an insufficient and inadequate a result, even with different sources. Trying different parameters during the experiment involving identifiers and operators does not lead to interesting results.

[1]Source: https://github.com/d-bingham/fse2017artifact

5.3 Bugram: Bug Detection with N-gram Language Models

5.3.1 Bugram presentation

Another scientific paper linked to this work concerns bugs detection called Bugram [41]. Even if many concepts presented do not directly refer here, there are some ideas under the same basis and especially, the token concept. It must be noted that recent studies demonstrated that n-gram language model [42] can spot or identify the regularities of software source code [43, 44].

Bugram takes advantage of n-gram language models instead of rules to detect bugs. This solution uses token sequentially exploiting the n-gram language model. Token sequences from the program are then evaluated depending on their probability in the learned model. Therefore, if there is a low probability in the learned model implies that the token sequence is a potential bug. In fact, a low probability token sequences may not indicate a bug but another problem. This latter involves bad programming practices, unusual uses of the code of which programmers may want to be aware.

Here, n-gram models learn probabilities of using a method depending on its context. With the learned probability distribution, they compute the possibility of each token sequence. Then, they report small probability token sequences as potential bugs.

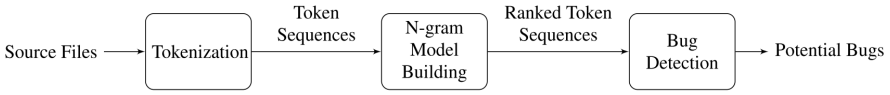

Figure 5.2: Tokenization in Bugram

As mentioned before, Bugram uses tokens when it comes to representing a program. Figure 5.2 shows the way how Bugram works. First, it converts an input into tokens. Then, it uses the tokens to build n-gram models. Finally, Bugram exploits the n-gram models to detect bugs. Without getting into the specifics of the n-gram model, the next section presents only the common concept to this work : the tokenization.

5.3.2 Bugram tokenization

In order to build a n-gram-model for a given project, a tokenization is needed of the source code. Here, the main challenge concerns the granularity: defining the length of the smallest element inside a system with great finesse is the key for success.

Based on existing works, a n-gram-model is built at the syntactic level. As

a result, tokens can be suggested for code completion and recommendations. For instance, listing 5.5 shows an incomplete "for loop". The suggestion for this example would be the token "i++".

```
1 for (int i=0; i<n;
```

Listing 5.5: Incomplete "for loop"

Unfortunately, this kind of n-gram model is useless at this level: nowadays, a basic integrated development environment or even a compiler will point out this kind of problem. Hence, a semantic level is more appropriate is the context of bug detection.

```
1 for (int i=0; i<n; i++) {
2         foo(i);
3 }
```

Listing 5.6: Complete "for loop"

At the semantic level, high-level tokens are selected. They represent the structure and context for the code source. As an example, listing 5.6 presents a simple "for loop" without any syntax errors. Bugram represents this part of the code with high level tokens as it can be seen on listing 5.7.

```
1 [<FOR>, foo(), <END_FOR>]
```

Listing 5.7: High-level tokens representation

The precision of bug detection is dependent on the control flow information. Therefore, Bugram adds the control flow elements into the token sets. These elements are method calls, constructors and initialisers:

- if/else
- for/do/while/foreach
- break/continue
- try/catch/finally
- return
- synchronized
- switch/case/default
- assert

Compared to the solution presented in this work, Bugram implements a higher view of tokens. In this master thesis, tokens are recognised one after another in a sequence but Bugram uses a semantic level with an abstract approach. The control flow elements from Bugram can be examined in contrast to different types of tokens presented as an add-on in section 3.2.2.1. Overall, Bugram proves its efficiency and paper results recommend using Bugram along side to existing rule-based bug detection.

Chapter 6

Future Works

This section presents future potential work concerning the subject. In fact many concepts were introduced in this master thesis and some of them should be discussed in greater detail. The purpose of this section is to suggest new avenues of investigation or even an alternative approach to the problem.

Many optimisations were done during the algorithm development but there is room for improvement. One of the key components is the mutation and crossover operators. Both have a set of pre-defined techniques creating new individuals based on variable parameters. The difficulty is that not every genetic programming operators are useful for a token sequence. Therefore, a study shall be conducted to see which operators produce valid and strong individuals. In other words, there is a possibility to create an interesting new population with genetic programming techniques by selecting proper parameters with the correct operators.

The tokenizer tool presented in this work is accessible and straightforward. With a syntactic approach, it performs a lexical analysis on a source code and tokens on output are stored inside a pattern structure as an ordered sequence. However, this tactic has its limit: the semantic aspect may be absent. For example, Bugram (section 5.3) shows interesting results with more abstract tokens. Exploring this way of proceeding may need to rethink the global structure of the algorithm and that is why it was not implemented in this work.

Alternatively, the token sequence method could be replaced by one of the other representations of the code. For instance, the abstract syntax tree is more often seen in the context of genetic programming. Additionally, the DEAP framework (section 2.3.7) has already a set of methods that deals with abstract syntax trees. Clearly, it is interesting to combine the vulnerable dataset and another source code representation in order to analyse the outcome with genetic programming techniques.

More generally, a study involving the nature of the results would be inter-

esting. In fact, in this current work, there is no behaviour analysis on the individuals. In other words, there is a list of strong mutants and offspring in the results but there is an in-depth analysis on token nature. This research study would lead to understand the vulnerability from a technical point of view.

Plus, it could also be worthwhile to see the context of a strong individual. As a reminder, the vulnerable dataset comes from the Linux kernel and the exact location of vulnerable files is well-known. Obviously, some files are more critical than others: a vulnerable mutant that affects a strategic file may be profitable for a further analysis. But then again, this idea concerns the behaviour of the individuals.

Chapter 7

Conclusion

This master thesis introduces generation of vulnerabilities with the aim of protecting software against unknown security issues. These undiscovered vulnerabilities are composed from existing source code using genetic programming techniques.

Starting from a vulnerable dataset based on the Linux kernel, a collection of well-known vulnerabilities were processed by external tools. In fact, the modified C-Tokenizer tool brought tokens for the lexical analysis and the evolutionary computation framework DEAP were responsible for the genetic programming part.

Clearly, results showed that a vulnerable pattern can be improved with genetic programming. In other words, the Darwin's principle of survival of the fittest works perfectly with vulnerabilities as they became stronger generation after generation.

Also, detailed investigation showed that there might be two kinds of powerful patterns on the outcome: generic and specific. The first one covers a lot of vulnerable files and the second one few. A further study may be considered in order to analyse the pattern behaviour.

The actual behaviour of a generated pattern, that is what is the *effect* of the vulnerability it introduces in the program, has not been explored in this thesis. Answering this question requires a different research method mixing automated and manual analysis. I nevertheless believe the work performed here set a basis for the automated generation of vulnerable patterns.

References

[1] Brian Marick. *The Craft of Software Testing: Subsystem Testing Including Object-based and Object-oriented Testing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995. ISBN 0-13-177411-5.

[2] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[3] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[4] Fabian Yamaguchi. Pattern-based vulnerability discovery. 2015.

[5] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 590–604. IEEE, 2014.

[6] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[7] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[8] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[9] Ivan Victor Krsul. *Software vulnerability analysis*. Purdue University West Lafayette, IN, 1998.

[10] Matt Bishop and David Bailey. A critical analysis of vulnerability taxonomies. *Cse-96-11*, (September 1996):0–14, 1996.

[11] Edward G. Amoroso. *Fundamentals of Computer Security Technology*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994. ISBN 0-13-108929-3.

[12] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. Software vulnerability discovery techniques: A survey. In *Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on*, pages 152–156. IEEE, 2012.

[13] William GJ Halfond, Shauvik Roy Choudhary, and Alessandro Orso. Penetration testing with improved input vector identification. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 346–355. IEEE, 2009.

[14] James Andrew Ozment. *Vulnerability discovery & software security*. PhD thesis, University of Cambridge, 2007.

[15] Stephanie Forrest. Genetic algorithms: principles of natural selection applied to computation. *Science*, 261(5123):872–878, 1993.

[16] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

[17] SN Sivanandam and SN Deepa. *Introduction to genetic algorithms*. Springer Science & Business Media, 2007.

[18] John R Woodward. Ga or gp? that is not the question. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 2, pages 1056–1063. IEEE, 2003.

[19] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[20] Pedro Larranaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[21] Jean-Yves Potvin. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research*, 63(3):337–370, 1996.

[22] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *ICGA*, 1987.

[23] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[24] Maarten Keijzer, Juan J Merelo, Gustavo Romero, and Marc Schoenauer. Evolving objects: A general purpose evolutionary computation library. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 231–242. Springer, 2001.

[25] Sean Luke. Ecj then and now. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1223–1230. ACM, 2017.

[26] Christian Gagné and Marc Parizeau. Open beagle: A new versatile c++ framework for evolutionary computation. In *GECCO Late Breaking Papers*, pages 161–168. Citeseer, 2002.

[27] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. An empirical analysis of vulnerabilities in openssl and the linux kernel. In *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*, pages 105–112. IEEE, 2016.

[28] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. Vulnerability prediction models: a case study on the linux kernel. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, pages 1–10. IEEE, 2016.

[29] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[30] Torben Ægidius Mogensen. *Basics of compiler design*. Torben Ægidius Mogensen, 2009.

[31] Christopher D. Manning and Hinrich Schütze. Foundations of statistical natural language processing. *Information Retrieval*, 4:80–81, 2001.

[32] James Halliday. C-tokenizer. URL `https://github.com/substack/c-tokenizer`.

[33] AK Santra and C Josephine Christy. Genetic algorithm and confusion matrix for document clustering. *International Journal of Computer Science*, 9(1):322–328, 2012.

[34] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78, 2012. ISSN 00010782. doi: 10.1145/2347736.2347755.

[35] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[36] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. ISSN 1089778X. doi: 10.1109/4235.996017.

[37] Yılmaz Kaya, Murat Uyar, et al. A novel crossover operator for genetic algorithms: Ring crossover. *arXiv preprint arXiv:1105.0355*, 2011.

[38] Xian-Huan Wen, Tina Yu, and Seong Lee. Coupling sequential-self calibration and genetic algorithms to integrate production data in geostatistical reservoir modeling. In *Geostatistics Banff 2004*, pages 691–701. Springer, 2005.

[39] Peter J Bentley and Jonathan P Wakefield. Hierarchical crossover in genetic algorithms. In *Proceedings of the 1st On-line Workshop on Soft Computing (WSC1)*, pages 37–42, 1996.

[40] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pages 511–522, 2017. doi: 10.1145/3106237.3106280.

[41] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: bug detection with n-gram language models. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 708–719, 2016. doi: 10.1145/2970276.2970341.

[42] Eugene Charniak. *Statistical language learning*. MIT press, 1996.

[43] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3.

[44] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884848.