THESIS / THÈSE

MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

Automatic Vulnerability Injection using Genetic Improvement and Static Code Analysers

BENIMEDOURENE, Charles

Award date:
2023

Awarding institution:
University of Namur

UNIVERSITÉ DE NAMUR
FACULTÉ D'INFORMATIQUE

UNIVERSITY OF NAMUR
Department of Computer Sciences
Academic Year 2022-2023

Automatic Vulnerability Injection using Genetic Improvement and Static Code Analysers

BENIMEDOURENE Charles

.... (Signature pour approbation du dépôt - REE art. 40)

Promotor : PERROUIN Gilles

Co-promotor : DEVROEY Xavier

Thesis presented for the obtention of a Master's diploma in Computer Sciences

With a specialisation in Software Engineering

Department of Computer Sciences – University of Namur

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Acknowledgement

First and foremost, I would like to thank Dr. Gilles Perrouin and Dr. Xavier Devroey for having given me the opportunity to work at the SnT in the context of my thesis. Furthermore, their guidance throughout the redaction of this thesis helped me to structure its content and make it more palatable to uninitiated readers, not familiar with the techniques and concepts I used.

I would also like to thank my supervisors at the SnT, namely Dr. Michail Papadakis and Dr. Renzo Degiovanni for their continued assistance and frequent feedback as well as for having directed me towards interesting frameworks and research papers to further my own thesis. These thanks extend to Dr. Matthieu Jimenez who introduced me to CodeQL and to the idea of using it in conjunction with genetic improvement. He offered interesting ideas as to what my thesis could be about, which I fully took advantage of.

Furthermore, I would like to mention how welcoming and agreeable to work with were the employees at the SnT and SerVal as well as the other interns, with whom I shared a common office. I'd like to particularly acknowledge the anecdotal yet precious advice of Guillaume Haben, be it when working or playing chess, and Aayush Garg for his expertise and guidance in using Vul4J.

An acknowledgement of the administrative and HR staff at the SnT, Unamur and Erasmus is also warranted as they made this trip possible and facilitated my temporary relocation. The internship was agreeable and allowed me to learn more about the practical applications of genetic improvement. I'm thankful for having had the opportunity to meet, talk and work with great and friendly people who didn't mind taking some of their time to provide needed guidance in getting started with my thesis.

Lastly, I would like to thank the unnamed employees who frequently fixed and refilled the coffee machines that were integral to the well-being of my productivity each morning.

Contents

List of Figures

List of Tables

Abstract/Résumé

Français

Cette thèse explore la possibilité d'appliquer du "genetic improvement" dans le but d'injecter des vulnérabilités dans des programmes. Générer des programmes vulnérables de la sorte permettrait d'automatiquement créer des sets de données, cruciaux dans l'entraînement de modèles de machine-learning dédiés à la détection des vulnérabilités. Cela permettrait d'améliorer grandement la sécurité logicielle des programmes en offrant aux programmeurs des outils mieux entraînés à détecter et à signaler les vulnérabilités.

Cette idée a été mise en pratique grâce à l'implémentation de VulGr, une modification d'un framework dédiée au genetic improvement nommé PyGGi. VulGr lui-même fait usage de CodeQL, un analyseur de code statique possédant une approche innovante dans la détection statique de vulnérabilités dans le but d'injecter des vulnérabilités dans des programmes du set de données Vul4J.

L'expérience s'est révélée infructueuse, CodeQL n'étant pas suffisamment précis et étant trop chronophage que pour produire des résultats concrets dans un laps de temps acceptable (moins de 72 heures). Cependant, l'approche présente un intérêt pour de futures applications, CodeQL étant un effort communautaire continu promettant de futures mises-à-jour ayant le potentiel de résoudre les problèmes mentionnés.

***Mots-clés** : genetic programming, CodeQL, Vul4J, vulnérabilités, détection de vulnérabilités, injection de vulnérabilités*

English

This thesis explores the idea of applying genetic improvement in the aim of injecting vulnerabilities into programs. Generating vulnerabilities automatically in this manner would allow creating datasets of vulnerable programs. This would, in turn, help training machine-learning models to detect vulnerabilities more efficiently.

This idea was put to the test by implementing VulGr, a modified version of the framework dedicated to genetic improvement named PyGGi. VulGr itself uses CodeQL, a static code analyser, offering a new approach to statical detection of vulnerabilities. VulGr's end goal was to use CodeQL to inject vulnerabilities into programs of the Vul4J dataset.

This experiment proved unsuccessful, CodeQL lacking accuracy and being too time-consuming to produce concrete results in an acceptable time span (less than 72 hours). However, the general approach and VulGr still retain their relevancy for future uses as CodeQL is an ongoing community effort promising new updates fixing the issues mentioned.

Keywords *: genetic programming, CodeQL, Vul4J, vulnerability, vulnerability detection, vulnerability injection*

Acronyms

1. **CWE** Common Weakness Enumeration
2. **GI** Genetic Improvement
3. **GP** Genetic Programming
4. **ML** Machine Learning
5. **PoV** Proof of Vulnerability

Chapter 1

Introduction

1.1 Context

This section provides a brief overview of the current context which stirred this thesis in the direction of using genetic improvement for the purpose of injecting vulnerabilities into non-vulnerable programs.

Cybersecurity threats

In the field of software and web development, cyber-attacks have become widespread, causing severe financial, reputational and legal damages to the victims of such attacks and their customers. One of these attacks is named Reflected[1] Cross Site Scripting[2] [14]. In this attack, the perpetrator finds a vulnerability that allows them to inject a script, for example through unsanitized web requests. This script can be maliciously used and triggered each time a user visits the infected website, usually to steal sensitive data, cookies and credentials.

Training machine learning models to detect vulnerabilities and the issue of data scarcity

Because of the damaging nature of such attacks, many techniques have been developed and experimented with in order to detect and prevent security vulnerabilities from being exploited by attackers. One of the main techniques currently being worked on, showing great promise, is based on machine-learning. Models are trained on large datasets of vulnerable programs and their fixed versions. Unfortunately, this technique is hampered by the lack of high-quality data, a phenomenon called data scarcity[14], as vulnerability detection models require large volumes of correctly labelled data for training. Furthermore, data scarcity is exacerbated by several factors, such as the complexity and diversity of modern software systems, the constantly evolving threat landscape, and the difficulties associated with collecting and labelling relevant data. As a result, researchers and practitioners face a significant challenge in developing effective machine learning-based approaches for vulnerability detection.

[1] https://portswigger.net/web-security/cross-site-scripting/reflected

[2] https://owasp.org/www-community/attacks/xss/

Using static code analysers to detect vulnerabilities

A competing technique in vulnerability detection is static code analysis. It is explained in detail in section 2.8 and is a less preferred alternative to machine learning vulnerability detection due to the rigidity of its rule-based implementation, causing static code analysers to exhibit unreliable behaviours when attempting to detect vulnerabilities in programs other than those they were specifically implemented for. This approach also historically required experts to implement the patterns static code analysers have to detect, making it a laborious and expensive operation to produce reliable and precise static code analysers.

However, CodeQL, a new static code analyser, has brighter prospects due to its novel approach to static code analysis and its community-based approach to writing rules for vulnerability detection. As it benefits from an open-source implementation and the backing of a wider community and GitHub, it is hoped that this static code analyser could overcome the challenges older static code analysers faced by allowing any user to write their own patterns for vulnerability detection and share them, thus making the process or implementing precise and reliable static code analysers less expensive and laborious.

The promising field of genetic programming and genetic improvement

Genetic Programming is a quickly evolving field of computer science focused on replicating the process of natural selection and emulating it to evolve programs in a chosen direction. It has produced interesting results in fields such as program repair [19] and program optimization [10]. It has become more popular in recent years, showing great promises for other purposes.

One of these emerging application of GP is in the field of cybersecurity. Although machine-learning is dominant in cybersecurity, especially when coming to vulnerability detection [12], it suffers from the previously explained scarcity of data.

Genetic Programming would not be used to compete against machine-learning vulnerability detection but to generate labelled vulnerable programs by “disimproving” non-vulnerable programs, making GP a potential solution to fixing data scarcity and a support for training ML models specialized in vulnerability detection.

1.2 Motivation of the research

There are many factors influencing the goals of this thesis. These motivating factors are summarized below and take inspirations from the issues and technologies mentioned in the context.

- **Solving the issue of Data Scarcity**

  Since data scarcity is a bottleneck slowing down the field of vulnerability detection through machine learning based approaches, finding a way to get more labelled and high quality vulnerable and non-vulnerable programs for the training of machine learning models could potentially help avoid future damaging security breaches.

- **Using Genetic Programming for the purpose of Vulnerability Injection**

  While genetic programming has been used to repair programs [11, 19], create SQL Injection attacks [3] and even to detect Cross Site Scripting attacks [1], no concrete and documented attempts have been made to apply GP for the purpose of injecting vulnerabilities in programs or in simpler terms: to create vulnerable programs.

- **Assessing whether CodeQL can deliver more reliable results**

  As previously mentioned, CodeQL possesses many interesting and novel attributes, making it a potentially attractive alternative to machine-learning vulnerability detection and a potentially superior implementation of the principles of static code analysis compared to other static code analysers. The potential appearance of better static code analysis tools could have a drastic and positive impact on the field of cybersecurity, simplifying and democratizing the capabilities of detecting vulnerable code.

For these reasons, the stated objective of this thesis is to investigate a new way of generating labelled high-quality data to address the issue of data scarcity. By using CodeQL and the concepts of genetic improvement, we hope to be able to build a framework capable of producing labelled vulnerable programs.

Chapter 2

State of the Art

In this chapter, we provide an overview of the latest researches on genetic improvement techniques and frameworks, as well as on fault injection.

2.1 Genetic Programming

Genetic Programming [16] is an evolutive approach aiming to create computer programs automatically. In order to achieve this, it mimics the process of natural selection, where organisms that are better at surviving (being "fitter") are likelier to survive and reproduce, in turn passing the genes that ensured their survival to the next generations of organisms, improving the specie's broader ability to survive generations after generations. On the contrary, harmful genes, negatively impacting the chances of survival, are less likely to be passed on, making them naturally disappear from the gene pool. New genes often appear through mutations, variations of genes producing new traits that can impact the fitness of organisms.

A well known example of natural selection is the giraffe. Giraffes that benefitted from the mutation of having longer necks got access to more resources to feed themselves as well as a better ability to spot threats from a distance, improving their chance of survival. On the other hand, giraffes with shorter necks could not find as many resources and could not spot predators as efficiently, causing that attribute to slowly disappear from the gene pool. This difference led to modern giraffes having longer necks. Other examples of this evolutive process are well documented, such as the differences in size and taste of modern tomatoes compared to their ancestors, that can be attributed to the fact humans tended to favour planting tomato trees yielding bigger and more nutritive tomatoes.

Genetic Programming emulates natural selection by automatically generating variants of the chosen program or source code by mutating it. These variants are called individuals and each individual is assessed according to a fitness function, assigning a fitness value to each individual. Whilst natural selection's goal is survival, genetic programming can have different goals, such as program optimization and repair. For each generation, a set of mutated individuals will be generated and evaluated by the fitness function, the evolutionary algorithm will then select some of these individuals to be the basis for the next generations of mutated individuals, thus enabling each consecutive generation to retain the pertinent and interesting mutations in the aim of attaining the desired result. The code can be compared to the genes and the fitness function to the survival chance.

2.2 Main components of a GP run

Applying Genetic Programming to a problem requires several decisions to be made, these decisions can be viewed as preparatory steps[16], these decisions, explained in greater detail below, are the following ones:

- Defining a terminal set.
- Defining a function set.
- Defining a fitness measure and function.
- Defining the parameters that will control the run.
- Defining a stopping/termination criterion.

2.2.1 Terminal Set

The terminal set consists of variables, constant values and even 0-arity functions (functions taking no parameters). When picturing an algorithm or program as a syntax tree, the leaves of such a tree would be the components of the terminal set. For example, the green nodes in the syntax tree (figure 2.1) are part of the terminal set of the if-then-else condition.

2.2.2 Function Set

The function set consists of arithmetic operations and all sorts of functions applied to elements of the terminal sets or other elements of the function set. The blue nodes in the syntax tree are members of the function set (figure 2.1).

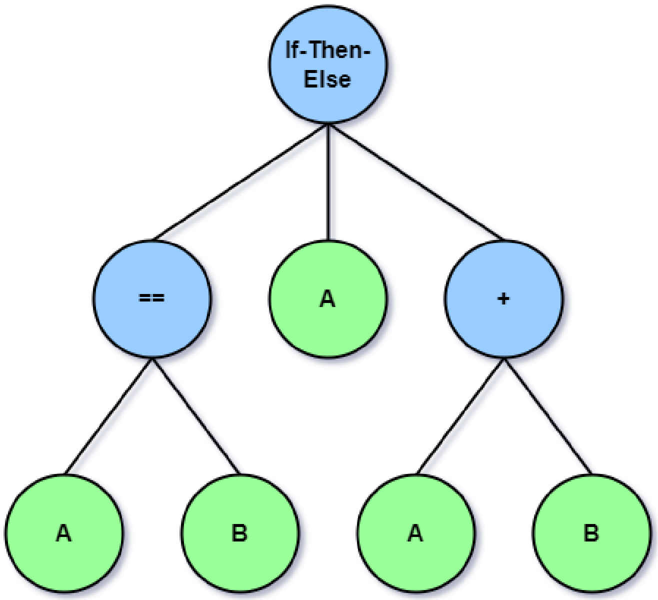

Figure 2.1: Syntax tree of an if-then-else condition
Figure inspired by figure 4 in [17]

2.2.3 Fitness Measure and Function

The fitness measure is a metric used to assess the quality, performance or the pertinence of an individual within a population, it enables one to quantify how well the specific individual achieves the goals that were set.

For example, how fast it runs, how close it is to achieve the goals that were set or even how well written the code is. During the evaluation phase of the genetic programming, each individual will be assigned a fitness value by the fitness function, a function dedicated to assessing the individual based on the set fitness measure. The fittest individuals will then be selected or at least have more chances of being selected as a basis for further mutations in the next generations (also named epochs), allowing the run to retain beneficial mutations.

Because of the sizeable overhead caused by the fitness function when it evaluates an individual, it is more common to use an interpreter to evaluate the evolved and mutated programs instead of compiling and building them.

2.2.4 Run-controlling Parameters

Multiple parameters control how the run behaves, the most common ones are the Population Size and the number of Generations, some Genetic Programming frameworks and runs also have parameters for the probability of performing genetic mutations, the weighting when selecting individuals to populate the next generation, the types of genetic mutations and for syntax tree-based mutations, a size and/or depth limit for the programs as the trees often grow to excessive sizes if left unchecked.

There is no established consensus on determining optimal and universal values assigned to these parameters, since determining the best values heavily relies on the implementation of the application and on the goals of the run, thus requiring experimentation and fine-tuning [16]. There is, however, an empirical method to defining and finding the optimal parameters values using Factorial Designs [13].

2.2.5 Termination Criterion

The termination criterion defines a stopping condition for the run. Typically, this criterion is reached once a solution to the problem, being a fit enough individual, is produced. If no viable solutions were produced, the termination criterion uses the maximum number of generations allowed as set in2.2.4 to define a stopping point for the run.

As an example, one could stop the run as soon as a solution is found (an individual achieving all the goals implemented in the fitness function), although some could also want to let it run for longer, until it reaches the defined maximum number of generations to get multiple solutions. If no solution is found before reaching the maximum number of generations, most genetic programming frameworks will return the best individual according to the fitness measure, some can return more individuals as well when supplementary data would be sought.

2.3 Selection

Selection is an integral part of Genetic Programming and makes use of the fitness function. This in an integral part of Genetic Programming, making use of the fitness function and sometimes randomness to select the individuals that will be chosen for reproduction and/or mutation.

The most common selection method is called tournament selection and consists of randomly selecting a number of individuals and comparing them, selecting the best one. This ensures that even marginally better individuals are selected yet keeps a certain level of diversity thanks to its random selection of the individuals that will be compared. It also automatically rescales the fitness, lowering the fitness pressure [1], since the individuals randomly chosen for the tournament are compared to each other only, not to fitter individuals from previous generations or from other tournaments. Automatic fitness rescaling thus helps to maintain diversity, allowing interesting and beneficial mutations, that would have been otherwise overshadowed by the fittest individuals not part of the tournament, to be kept in the genetic pool.

2.4 Mutations

Mutations are genetic operators employed in evolutionary algorithms, such as Genetic Programming, that introduces genetic variability and enables exploration within the population of individuals by randomly mutating parts of the program. It is a crucial component of the evolutionary process, mimicking the biological concept of genetic mutation.

In the context of Genetic Programming, mutation operates on the tree-based representation of individuals, where each node in the tree corresponds to a member of the function set or terminal set. The process of mutation involves making small modifications to the genetic material (i.e., the program code) of an individual, thereby creating a slightly altered version of the original solution.

[1] Selection Pressure is a term designating how biased towards the fittest individuals the selection is. A strong selection pressure favours the fittest individuals whilst a weak selection pressure favours more diversity.

2.4.1 Subtree Mutation

The subtree mutation randomly replaces a subtree, ranging from a single node to a larger substructure of multiple nodes, with a new randomly generated subtree. Such a mutation is represented can be seen in figure 2.2.

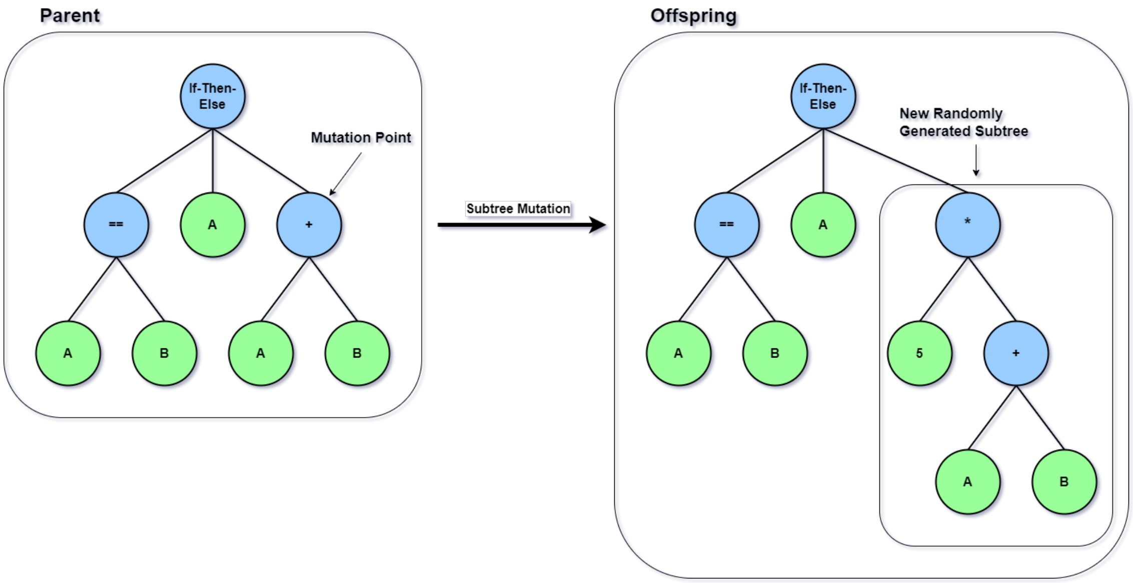

Figure 2.2: Example of a subtree mutation

2.4.2 Node Replacement Mutation

Akin to the subtree mutation, the node replacement mutation replaces a node of the tree with a randomly generated new one, retaining some characteristics such as the type or number of arguments to ensure the replacement still leads to a valid program.

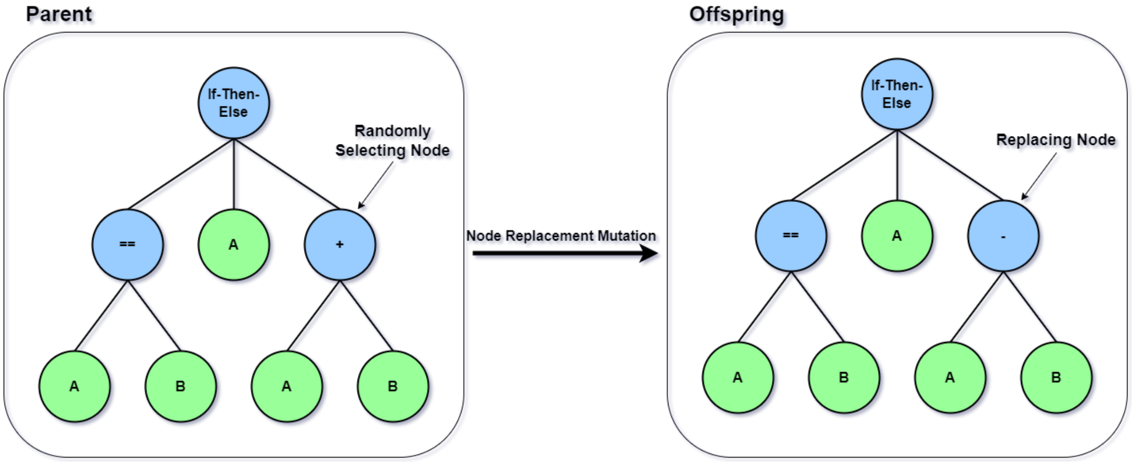

Figure 2.3: Example of a node replacement mutation

2.4.3 Shrink Mutation

Shrink mutation removes a randomly chosen subtree and replaces it with a randomly created terminal. This mutation operator is often used to reduce the size of the program.

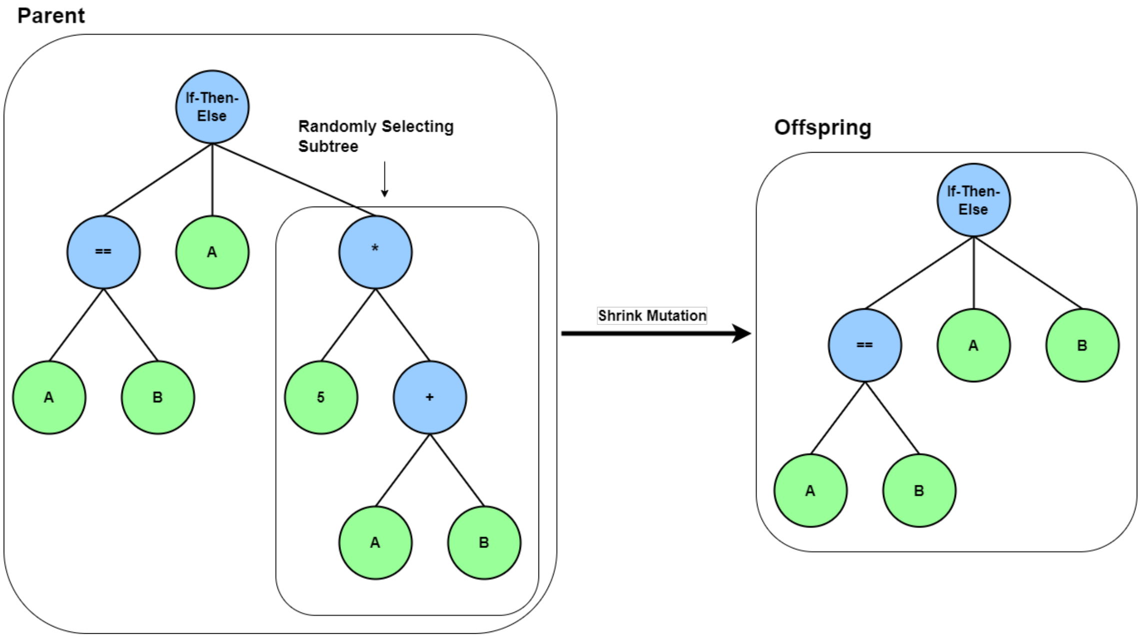

Figure 2.4: Example of a shrink mutation

2.4.4 Hoist Mutation

Similarly to the shrink mutation, hoist mutations aims to reduce the size of the program. It accomplishes this goal by reducing the individual to a copy of a subtree of its parent.

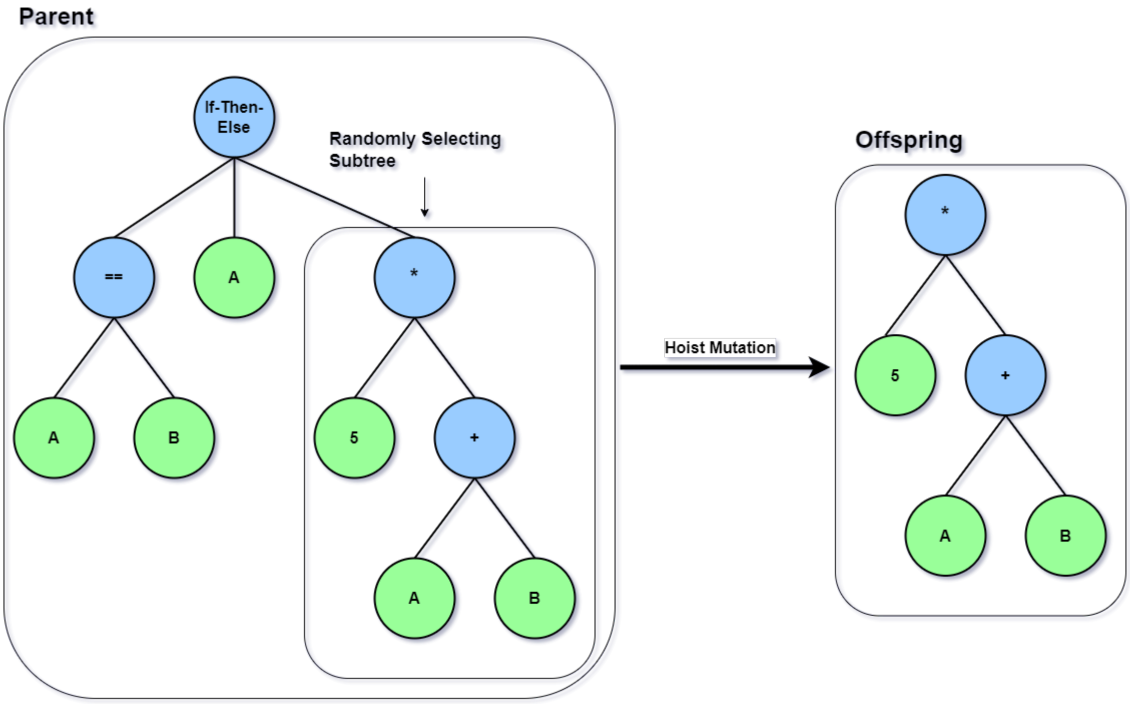

Figure 2.5: Example of a hoist mutation

2.4.5 Line Mutations

Line mutations do not fully make use of the syntax tree, sometimes too complex and cumbersome depending on the nature of the code being modified and the expected outcome, and instead randomly deletes, replaces or adds lines in the existing program. These mutations can be more advantageous in certain situations as they enable more granularity, reduce the search space and causes the individuals resulting of such mutations to retain their code structure. As such, these mutation operators are often used in program repairs and on larger projects.

2.5 Crossover

Crossover is a genetic operator that combines individuals from previous generations to produce new original individuals. This process emulates biological reproduction, where the DNA of the two parents are mixed. In the case of Genetic Programming, branches of the syntax tree representing the program are switched and combined to produce the new original program retaining characteristics from its parent programs. Figure 2.6 represents such a crossover.

There are two families of crossover operators, the homologous and non-homologous, traditional, crossover. Traditional, non-homologous crossover differs from biological reproduction due to the possibility of a subtree being moved to a totally different location in the new offspring tree. Homologous crossover operators, on the contrary, retain the location of previous genetic material.

Crossover can be split into three main steps:

- Selecting parent individuals

  Selection of individuals from previous generations is mandatory and the most important step of the process. This selection process is explained in further detail in section 2.3.

- Identifying crossover points within the selected parents.

  Once parent individuals are selected, points within each of the parents have to be identified. Although these are typically chosen randomly, some specific crossover techniques can add constraints to this randomness, confining the selection of the crossover points to look for a common point present in a common region that has the same arity or function sets. These points are the locations where the exchange of genetic material, effectively subtrees, will take place.

- Exchanging the genetic material and generating the offspring.

  At the crossover points, subtrees or substructures from the parent individuals are swapped to create new offspring individuals. This exchange involves copying the selected subtree from one parent to the corresponding location in the other parent, and vice versa, generating two new altered copies.

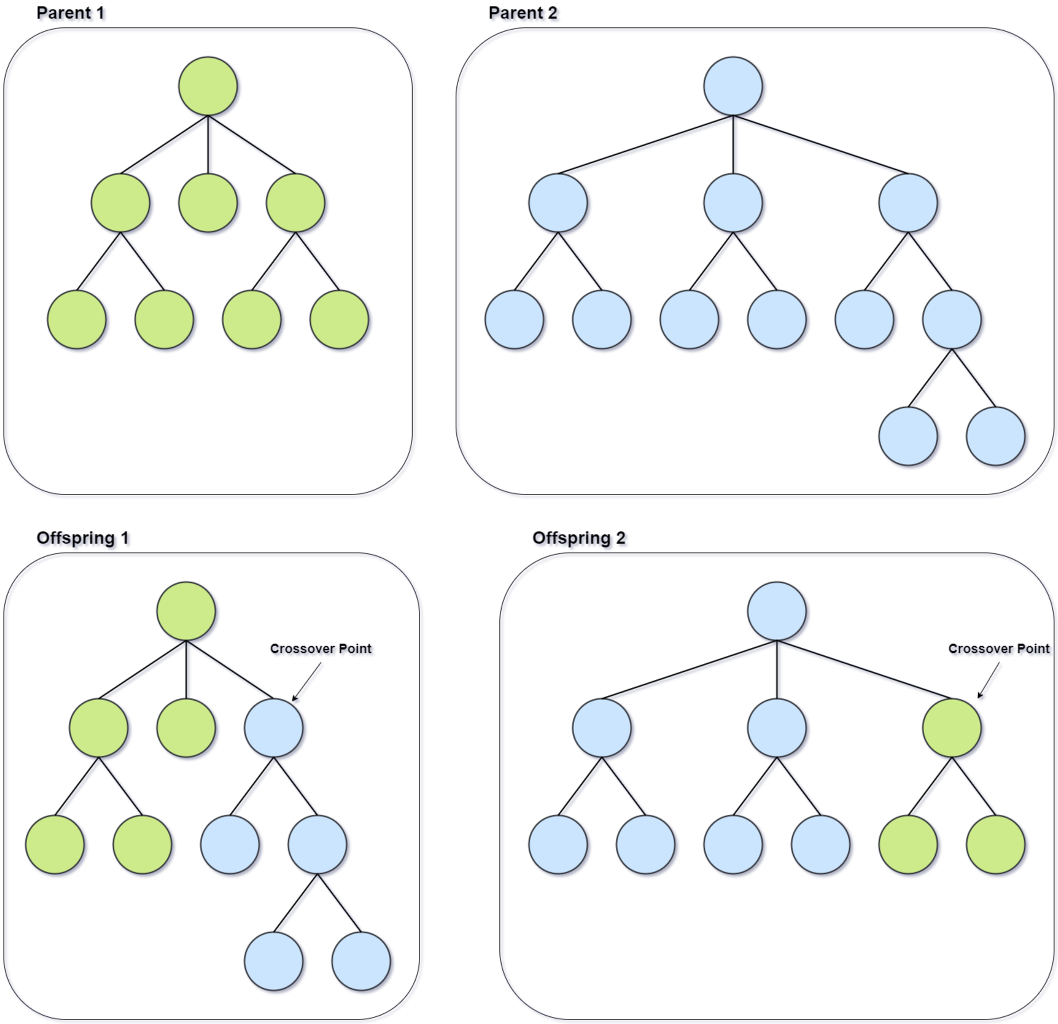

Figure 2.6: Example of a one-point Crossover

2.6 Genetic Improvement

Genetic improvement of programs is the application of the principles of genetic programming on a software or system, typically to improve its non-functional properties such as reducing its resource cost or optimizing its performance while retaining its functional results, effectively improving it [10]. Genetic improvement thus naturally incorporates aspects of genetic programming such as the need for a fitness function, a termination criterion and makes use of the same genetic operators, applying mutations and crossover reproduction.

2.7 Datasets

Datasets Selecting a dataset on which to operate the genetic improvement in order to inject vulnerabilities is a crucial part of the process and has an important effect on the end results. Multiple types of datasets exist

2.8 Static Code Analysis

Static code analysis is a technique used to analyse the source code of a project, without actually executing it. It is most commonly used for identifying syntax errors, code smells, issues and even security vulnerabilities. Contrary to dynamic analysis, which involves executing the code to observe its behaviour, static analysis examines the code's syntax, structure and semantics.

In the field of cyber-security, static code analysis is rarely used alone by itself, since it has been known to have limitations in this regard [9], such as being unable to catch all vulnerabilities or being prone to find false positives (code incorrectly flagged as being vulnerable) or false negatives (vulnerable code not identified as such by the static code analysis tool).

There are many static code analysis tools, ranging from linters, aiming to improve the quality of a program by analysing its source code, effectively enforcing consistency and good practices, to CodeQL, allowing identification of security vulnerabilities and code smells by running queries against a database of the source code. These differ not only in their goals but also in their implementation and functioning,

2.8.1 CodeQL

CodeQL is an open source and free for research static analysis tool focusing on semantics, currently owned by GitHub (Microsoft). The tool enables users to write queries, similar to SQL queries in their syntax and results, that will then be run against a database of the source code. This database is constituted by analysing the source code and by additionally observing the compilation process for compiled languages such as Java. Using CodeQL can be decomposed into four steps:

- Generating a CodeQL database of the code
- Writing/Finding CodeQL queries
- Running the queries
- Interpreting the results

Queries

CodeQL's queries allow for users to use a SQL-like approach to finding specific patterns in the source code of a program. A basic CodeQL query is structured in an almost identical fashion to its SQL counterpart, starting with the "from" keyword, then the conditional section with the "where" keyword and ending with the "select" keyword. Modules and libraries can also be imported at the beginning of a query by using the "import" keyword. One template for a typical CodeQL query is depicted in figure 2.7.

CodeQL is natively packaged with a multitude of these queries, made by GitHub researchers and by the community. CodeQL's native queries cover a wide range of uses for each supported language and even for some frameworks. They are often used for code quality assessment, performance and complexity, code architecture, compatibility and security.

Security queries all follow the same structure when it comes to their documentation and implementation. The .ql file contains a short description of the CWE vulnerability and a precision rating, going from low to high, as shown in figure 2.8. High precision queries are less prone to false positives and true negatives, contrary to low precision queries.

Each security query also has a .qlhelp file providing more detail for the specific CWE vulnerability it aims to detect, recommendations to avoid implementing such vulnerabilities and examples of vulnerable code.

```
/**
 *
 * Query metadata
 *
 */

import /* ... CodeQL libraries or modules ... */

/* ... Optional, define CodeQL classes and predicates ... */

from /* ... variable declarations ... */
where /* ... logical formula ... */
select /* ... expressions ... */
```

Figure 2.7: Template for a CodeQL query, copied from the official documentation

```
/**
 * @name Overly permissive regular expression range
 * @description Overly permissive regular expression ranges match a wider range of
     characters than intended.
 *              This may allow an attacker to bypass a filter or sanitizer.
 * @kind problem
 * @problem.severity warning
 * @security-severity 5.0
 * @precision high
 * @id java/overly-large-range
 * @tags correctness
 *       security
 *       external/cwe/cwe-020
 */

private import semmle.code.java.regex.RegexTreeView::RegexTreeView as TreeView
import codeql.regex.OverlyLargeRangeQuery::Make<TreeView>

TreeView::RegExpCharacterClass potentialMisparsedCharClass() {
  // nested char classes are currently misparsed
  result.getAChild().(TreeView::RegExpNormalChar).getValue() = "["
}

from TreeView::RegExpCharacterRange range, string reason
where
  problem(range, reason) and
  not range.getParent() = potentialMisparsedCharClass()
select range, "Suspicious character range that " + reason + "."
```

Figure 2.8: Source code of the security query made for CWE-20

Chapter 3

Research Questions

Now that we have had the opportunity to list current state-of-the-art techniques and technologies as well as have provided an introduction as to the motives and goals of this thesis, we can formulate the main research questions that we will try to find answers to.

- **Question 1: How reliable is CodeQL compared to previously tested static code analysers when dealing with security vulnerabilities?**

  This question steams from the proven inaccuracy and unreliability of older Static Code Analysers [9]. Since we also intend to use CodeQL and its community-made security queries to assess the fitness of potentially vulnerable mutants of Vul4J programs (as explained below in section 4), finding an answer to the question will require us to run the GI process using CodeQL and running CodeQL's security queries against Vul4J's vulnerable programs.

- **Question 2: How can CodeQL's take on source code analysis through databases of code be realistically used for the purpose of GI?**

  As genetically improving a program through random mutations of its source code is a lengthy process involving thousands of mutated variants of the source code and other variants, the performance cost of rebuilding databases of each of these mutated versions of the source code is a real concern and potential bottleneck in applying GI using CodeQL. Running a suite of queries for each of these mutated individuals could also to be time-consuming as well.

- **Question 3: How can genetic improvement be used in the field of automatic vulnerability injection?**

  Due to the high diversity and complexity of the implementations of potentially vulnerable programs, automatically injecting vulnerabilities through genetic improvement poses the risk to be proved to be overly ambitious for modern hardware, current mutation operators and for GI selection algorithms. It is thus important to assess the current feasibility of such an application of GI or its future uses.

Chapter 4

Methodology

4.1 Vulnerability Injection using Genetic Programming

Whilst genetic programming is oft-used for program repairs and optimization, it has never been used to generate faults in a functioning program. Instead of starting from a deficient program that has to be repaired, the idea is to begin with an already fixed or non-vulnerable program and to derivate vulnerable mutations, allowing us to improve the dataset of existing vulnerabilities.

The approach to this use of genetic improvement to inject vulnerabilities into a program follows an iterative process, as represented in figure 4.1. The idea can be summarized as such:

- **Step 1: Finding a "parent" non-vulnerable program (dataset)**

  The non-vulnerable program should have the potential to become vulnerable. Fixed/repaired programs are thus a good starting point, since they were initially vulnerable. This program will serve as a parent on which the genetic improvement will be applied.

- **Step 2: Generating mutants offspring**

  Once a suitable program is found, the genetic improvement run can start. The first step of such a run is to generate mutated individuals to compose a generation. The number of mutated individuals to create is determined by the size of the generation, as defined in 2.2.4.

- **Step 3: Analysing/assessing the fitness of the mutants**

  The mutants are assessed according to the fitness function (explained in 2.2.3) and are each assigned a fitness value. The fitness function makes use of a static code analysis tool in order to detect traces of a potential future vulnerability. The code doesn't need to be vulnerable per se, but closer to become vulnerable in order to get a higher fitness value.

- **Step 4: Checking if the stopping criterion has been met**

  The run can stop if the stopping criterion has been met, such a stopping criterion would ideally be a mutated individual having the maximal fitness value, effectively meaning the static code analysis tool considers it vulnerable.

  If no such individuals have been created during the process, the stopping criterion is met once the maximum number of generations/iterations has been reached.

- **Step 5: Selecting individuals for reproduction**

  If the stopping criterion hasn't been reached yet, we select the individuals to be chosen for reproduction for the next generation, coming back to step 2 of the process but hopefully starting from a parent that is closer to being vulnerable.

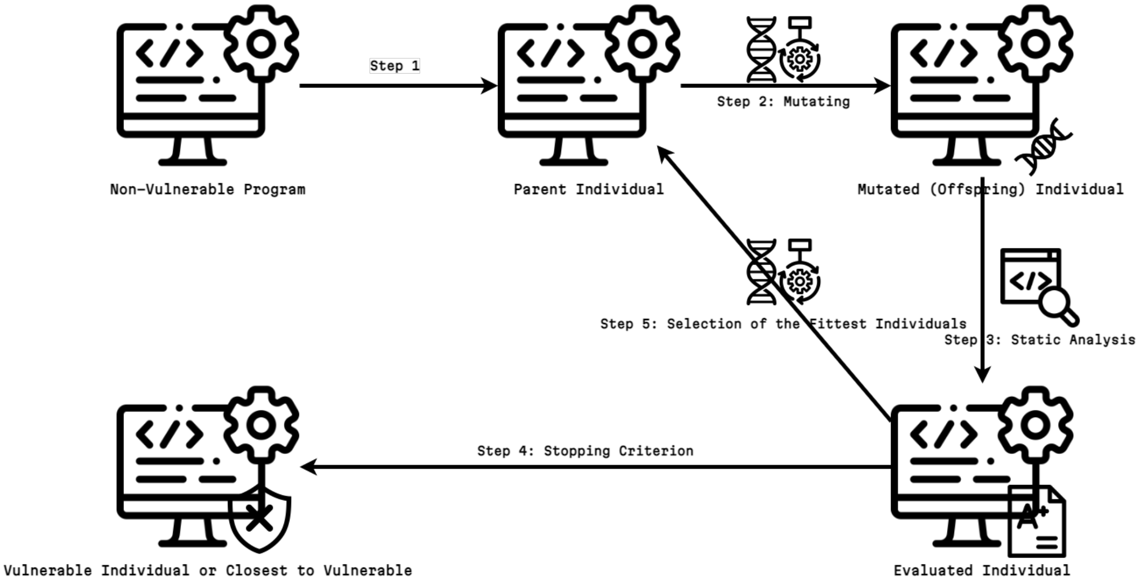
Step 1
Step 2: Mutating
Non-Vulnerable Program
Parent Individual
Mutated (Offspring) Individual
Step 5: Selection of the Fittest Individuals
Step 3: Static Analysis
Step 4: Stopping Criterion
Vulnerable Individual or Closest to Vulnerable
Evaluated Individual

Figure 4.1: Iterative Process of the Methodology

4.2 Dataset

Selecting a good dataset (non-vulnerable program) is a crucial part of the process, influencing all the next steps in our goal to apply genetic improvement successfully and inject a vulnerability into the selected dataset.

The first factor constraining the possibilities of which program can be used as our initial dataset was the selection of a programming language. This choice was made after we selected PyGGi as our genetic improvement framework, as explained below in section 4.3.1. Whilst PyGGi can handle any language thanks to srcML, its main implementation is particularly suited to Java, C and Python. For reasons similar to those mentioned page 13 in [15], we decided to focus on Java.

There are many suitable Java programs that could have been chosen, such as the Juliet Test Suite for Java, some of the SARD's test programs [4], SAMATE [8], OWASP and Vul4J. Vul4J seemed to be the more natural choice since it is lightweight, geared toward repairs and thus contains fixed versions of all the vulnerabilities and focuses on CWE vulnerabilities which corresponded to our static code analyser's queries as explained in section 4.4.1 and for these reasons we decided to proceed with it.

4.2.1 Vul4J

Vul4J is a dataset containing real-world vulnerabilities [5]. Each vulnerability in the dataset has a CWE ID attached to it, rendering it simpler to find the vulnerabilities we would like to work with and the corresponding CodeQL query to apply on it. The tables mapping the Vul4J vulnerabilities to CWE vulnerabilities are available in the annex, tables A.4 and A.5.

Moreover, Vul4J possesses many other interesting characteristics, such as providing a human-made patch to the vulnerability, a test suite able to detect whether the vulnerability is still present in a program of the dataset (thus acting as a proof-of-vulnerability (PoV)) and simple to use commands to easily compile and test said programs.

The human-made patches that repair the vulnerabilities can be used as our initial individuals on which to apply genetic improvement until they become vulnerable, providing us with many non-vulnerable programs that exhibit great potential to become vulnerable, as well as allowing us to exactly know which vulnerability we should work towards. The PoV can additionally provide a second layer of certitude when assessing the presence of a vulnerability after mutation, as static code analysers are known for having false positives and true negatives [9].

4.3 Genetic Improvement Framework

Genetic improvement frameworks are libraries, software systems or platforms laying the groundwork and providing the infrastructure and tools for the purpose of applying the genetic improvement process on a program. They have pre-implemented methods and classes that can be easily extended and overridden to tailor their behaviour to the goals of the user and to the specifics of the programs to improve.

These pre-made frameworks dedicated to genetic improvement can substantially simplify our implementation of the genetic improvement process as well as enforce standards and common practices, making sure no steps are missed and that the most efficient and common GP/GI techniques are used. For this purpose, multiple frameworks were investigated, and we chose to settle on the Python General Framework for Genetic Improvement (PyGGi) [2], a lightweight framework written in Python.

4.3.1 PyGGi

The Python General Framework for Genetic Improvement offers many interesting functionalities. Not only is it lightweight, it can also technically support any language thanks to the srcML engine. In our case it is useful for its proven application on Java source code as some sample examples provided with it operate automated, genetic program repair on Java scripts.

PyGGi's use of an XML-based intermediate representation of the source code also allows genetic operators to be used on it instead of the source code, in turn making it possible to create universal genetic operators that will work on any language once converted to the XML-based representation using srcML.

When looking at the steps explicated in section 4.1, PyGGi natively handles step 2 to 5, mutant generation as well as individual selection being already implemented, automatic analysis only requiring us to write our own fitness function and at the same time defining our own stopping criterion.

4.3.2 VulGr

VulGr (Vulnerability Generation) is our modified extension of PyGGi, tailored for the purpose of Genetic Improvement in the aim of generating vulnerabilities in non-vulnerable programs. It is publicly available on GitHub[7]. The code in the repository itself doesn't need to be run locally, as it was made with Google's Colaboratory in mind and is used by our Colaboratory[6].

VulGr's main implementation is located in vulgr/vulgr_testing.py and an unfinished and more integrated one can be found in example/vul4j_unfix.py. The vulgr_testing script runs a single time, executing the fitness function on the program. The execution can be summarized as creating a CodeQL database of the program, running our CodeQL subqueries (explained in 4.4.1) against the previously generated database, then reading the output logs of each query to assess how many of them were triggered.

4.4 Static Code Analyser

Static Code Analysis is explained in greater detail in section 2.8. As one of the overall goals of this experimentation was to try CodeQL, GitHub's more recent static code analyser, we didn't judge it necessary for us to explore other static code analysers as they have been proven to have limited reliability and shortcomings when it comes to vulnerability detection [9].

4.4.1 CodeQL

As we already delved into CodeQL's inner workings in section 2.8.1, this section focuses on which security queries were tested, what results they yielded and how we used them to write new subqueries.

Security Queries

Security queries are the most promising queries offered by CodeQL in our aim to inject vulnerabilities. They all are related to vulnerabilities listed in the Common Weakness Enumeration (CWE) which is helpful in finding the corresponding Vul4J repaired program and enables us to already know which specific queries to run on a specific Vul4J vulnerability.

The expected workflow consists of exploring the mapping of each Vul4J vulnerability to its CWE weakness as provided in tables A.4 and A.5 and matching it against our list of security queries in hopes of finding programs whose vulnerable versions will trigger the CodeQL query dedicated to the same CWE weakness. That Vul4J vulnerability would then become the parent program that will be mutated, as explained in step 1 of 4.1. The execution of this process is detailed in section 5.1.

Subqueries

Once a parent Vul4J program with a corresponding CodeQL query able to detect its vulnerability is found and that mutants based on the source code of its fixed version are generated by using PyGGi (effectively applying step 2 of 4.1), we can start implementing step 3, namely creating a fitness function using the static code analyser CodeQL.

As we know that the initial CWE related CodeQL query can detect the vulnerability and in turn allows us to infer that the initial program is vulnerable, our emerging idea became to split that specific query into more granular subqueries, allowing them to detect finer causes for the appearance of such a vulnerability or in other terms, if the program is closer to becoming vulnerable or not.

For example, let's define a program VP as being the vulnerable Vul4J program (fitness value of 1), RP the repaired version of VP and Q the CodeQL query able to detect the vulnerability in VP. Now let Q be split into Q1, Q2, Q3 and Q4. In our example, RP only triggers Q1 out of the 4 subqueries, allowing us to value its fitness as being 0.25. If we now mutate RP and name its resulting children as MP, MP might trigger Q1 and Q2 allowing us to value its fitness as being 0.5, much fitter than its parent and closer to becoming vulnerable. This example is illustrated in figure 4.2.

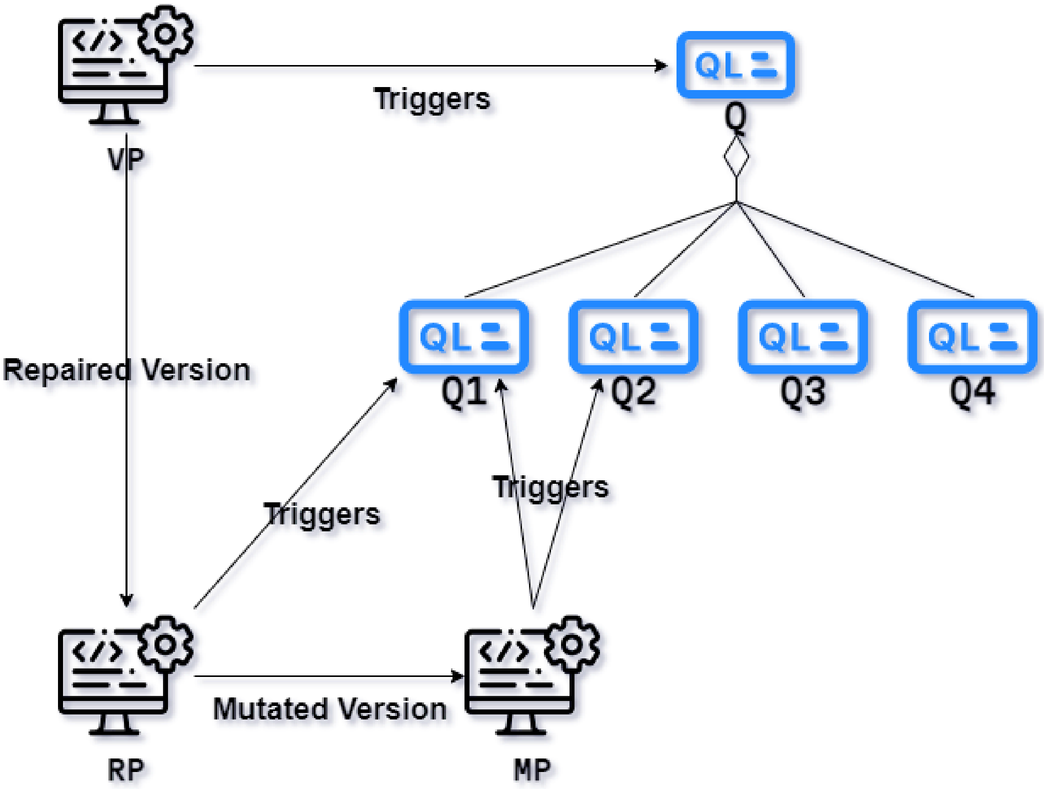

Figure 4.2: Use of Subqueries for the Fitness Function

4.5 Concrete Steps

The theoretical steps explored in section 4.1 can now be concretized and described as follows:

- **Step 1: Finding a "parent" non-vulnerable program in the Vul4J dataset**

  Finding a suitable Vul4J program can be summarized as finding a Vul4J vulnerable program whose vulnerability can be detected by the corresponding CodeQL query and having a fixed, repaired version in which to inject vulnerabilities.

- **Step 2: Generating mutants offspring**

  Mutant generation will be handled by PyGGi, offering us the ability to natively choose between line and statement mutations, as well as allowing us to easily add our own mutation operators if needed.

- **Step 3: Analysing/assessing the fitness of the mutants**

  This step will also be handled by PyGGi, making use of CodeQL. A custom fitness function will be implemented in PyGGi to execute CodeQL's database creation process (for the source code of the mutated individual) and to execute CodeQL's queries on a database of said source code. The logs of these executions will then be automatically analysed to assess how many subqueries were triggered by reading and looking for specific keywords and values in them.

  Subqueries will be manually implemented by looking at the code of the initial query that was able to detect the vulnerability in the program selected in step 1 and splitting it according to elements such as the "OR" and "AND" logical operators.

- **Step 4: Checking if the stopping criterion has been met**

  This step is included in step 3's rewriting of the fitness function, ensuring the runs stop once a vulnerable program is found by the subqueries. Additionally, PyGGi already covers the case where no completely fit mutant is generated before the user defined maximum number of generations has been reached.

- **Step 5: Selecting individuals for reproduction**

  Once again, this step is natively handled by PyGGi which looks at the fittest individuals from previous generations to select individuals for reproduction.

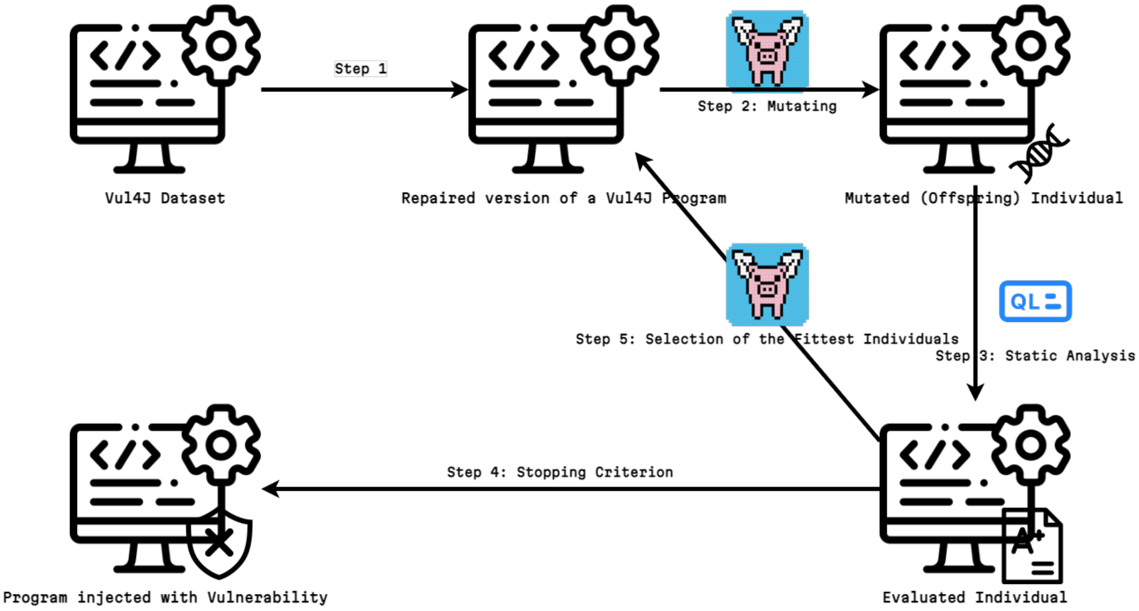
Step 1
Step 2: Mutating
Vul4J Dataset
Repaired version of a Vul4J Program
Mutated (Offspring) Individual
QL
Step 5: Selection of the Fittest Individuals
Step 3: Static Analysis
Step 4: Stopping Criterion
Program injected with Vulnerability
Evaluated Individual

Figure 4.3: Concrete Iterative Process

Chapter 5

Experimentation

This chapter delves into the application of the ideas and processes explained in chapter 4.

5.1 Finding a suitable Vul4J program

Part of step 1 in 4 was to find a dataset, we chose Vul4J as explained in 4.2. We now needed a repaired version of a specific program of this dataset to run the GI process on. Following our previously described methodology, we started trying the firsts Vul4J programs being attached to a CWE weakness, which involved having CodeQL build a database of the program by monitoring its compilation then running the CodeQL query corresponding to the CWE weakness against it. The results of running the queries against the programs can be found in table 5.2.

As can be seen in the table, Vul4J-34 yielded interesting results contrary to all the previous attempts, prompting us to proceed to the next steps with it as our initial parent program/dataset on which to apply GI. We also did not run the queries contained in files AndroidWebViewAddJavascriptInterface.ql and AndroidWebViewSettingsEnabledJavaScript.ql as in the mapping available in the annex (A.4), it is detailed this specific Vul4J vulnerability is linked to CWE-079, Cross-site Scripting and not other CWE-079 queries.

Table 5.1 lists the results of running 36 CodeQL security queries on the first 25 Vul4J programs having vulnerabilities listed in CWE. Out of 36 queries run, only one was able to reliably find a vulnerability. It can be noted that the query compilation time is a step that is automatically skipped if said query was already used before and cached by CodeQL. This effectively translates in a shorter total run time when using them for GI as the queries will only be compiled once for the first individual. The 36 tests were run and timed on Google Colaboratory, running them on more powerful hardware would reduce the time required to execute them. Hardware availability and performance is also subject to change when using Colaboratory without a paid subscription.

Number of Vulnerable Programs Tested	False Negatives	True Positives	Average Vul4J Initialization Time
25	24	1	<1s
Average CodeQL Database Creation Time	Average Query Compilation Time	Average Query Execution Time	Average Total Run Time
86.3s	182.1s	41.8s	310.2s

Table 5.1: Results of testing 25 Vul4-J vulnerabilities with CodeQL security queries

Vul4J Name/ID	Query Folder/CWE ID	Query File	Result
Vul4J-1	CWE-020	ExternalAPIsUsedWithUntrustedData.ql	None
Vul4J-1	CWE-020	OverlyLargeRange.ql	None
Vul4J-1	CWE-020	UntrustedDataToExternalAPI.ql	None
Vul4J-2	CWE-611	XXE.ql	None
Vul4J-2	CWE-611	XXELocal.ql	None
Vul4J-3	None	None	None
Vul4J-4	None	None	None
Vul4J-5	None	None	None
Vul4J-6	CWE-835	LockOrderInconsistency.ql	None
Vul4J-7	CWE-835	LockOrderInconsistency.ql	None
Vul4J-8	CWE-835	LockOrderInconsistency.ql	None
Vul4J-9	None	None	None
Vul4J-10	CWE-020	ExternalAPIsUsedWithUntrustedData.ql	None
Vul4J-10	CWE-020	OverlyLargeRange.ql	None
Vul4J-10	CWE-020	UntrustedDataToExternalAPI.ql	None
Vul4J-11	CWE-264	None	None
Vul4J-12	CWE-835	LockOrderInconsistency.ql	None
Vul4J-13	CWE-835	LockOrderInconsistency.ql	None
Vul4J-14	CWE-020	ExternalAPIsUsedWithUntrustedData.ql	None
Vul4J-14	CWE-020	OverlyLargeRange.ql	None
Vul4J-14	CWE-020	UntrustedDataToExternalAPI.ql	None
Vul4J-15	CWE-611	XXE.ql	None
Vul4J-15	CWE-611	XXELocal.ql	None
Vul4J-16	CWE-264	None	None
Vul4J-17	None	None	None
Vul4J-18	CWE-22	TaintedPath.ql	None
Vul4J-18	CWE-22	TaintedPathLocal.ql	None
Vul4J-18	CWE-22	ZipSlip.ql	None
Vul4J-19	None	None	None
Vul4J-20	None	None	None
Vul4J-21	CWE-254	None	None
Vul4J-22	CWE-284	None	None
Vul4J-23	CWE-079	AndroidWebViewAddJavascriptInterface.ql	None
Vul4J-23	CWE-079	AndroidWebViewSettingsEnabledJavaScript.ql	None
Vul4J-23	CWE-079	XSS.ql	None
Vul4J-23	CWE-079	XSSLocal.ql	None
Vul4J-24	CWE-611	XXE.ql	None
Vul4J-24	CWE-611	XXELocal.ql	None
Vul4J-25	CWE-079	XSS.ql	None
Vul4J-25	CWE-079	XSSLocal.ql	None
Vul4J-26	CWE-020	ExternalAPIsUsedWithUntrustedData.ql	None
Vul4J-26	CWE-020	OverlyLargeRange.ql	None
Vul4J-26	CWE-020	UntrustedDataToExternalAPI.ql	None
Vul4J-27	CWE-264	None	None
Vul4J-28	CWE-264	None	None
Vul4J-29	CWE-264	None	None
Vul4J-30	CWE-020	ExternalAPIsUsedWithUntrustedData.ql	None
Vul4J-30	CWE-020	OverlyLargeRange.ql	None
Vul4J-30	CWE-020	UntrustedDataToExternalAPI.ql	None
Vul4J-31	None	None	None
Vul4J-32	None	None	None
Vul4J-33	CWE-077	None	None
Vul4J-34	CWE-079	XSS.ql	Detected

Table 5.2: Results of running CodeQL queries on Vul4J programs

5.2 Creating the subqueries

Now that we have isolated Vul4J-34 as a vulnerable program, able to be detected by the query related to CWE-079, we were able to start working on splitting the query into subqueries as theorized in 4.4.1. The source code of the query is shown in figure 5.1.

```
import java
import semmle.code.java.dataflow.FlowSources
import semmle.code.java.security.XSS
import DataFlow::PathGraph

class XssConfig extends TaintTracking::Configuration {
 XssConfig() { this = "XSSConfig" }

 override predicate isSource(DataFlow::Node source) { source instanceof
    RemoteFlowSource }

 override predicate isSink(DataFlow::Node sink) { sink instanceof XssSink }

 override predicate isSanitizer(DataFlow::Node node) { node instanceof
    XssSanitizer }

 override predicate isSanitizerOut(DataFlow::Node node) { node instanceof
    XssSinkBarrier }

 override predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::Node
    node2) {
  any(XssAdditionalTaintStep s).step(node1, node2)
 }
}

from DataFlow::PathNode source, DataFlow::PathNode sink, XssConfig conf
where conf.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "Cross-site scripting vulnerability due to a
   $@.",
 source.getNode(), "user-provided value"
```

Figure 5.1: Source code of the security query for Cross Site Scripting (XSS - CWE-079)

Since the query makes use of other methods and classes implemented in different files and because these other methods and classes themselves extend other parent classes, displaying the entire source code for this query wouldn't help much in comprehending it.

For the purpose of clarity, the code source of the resulting subqueries we made after splitting the XSS.ql can be found in the annex A.3 and figure 5.2 displays their respective relationships to the original query.

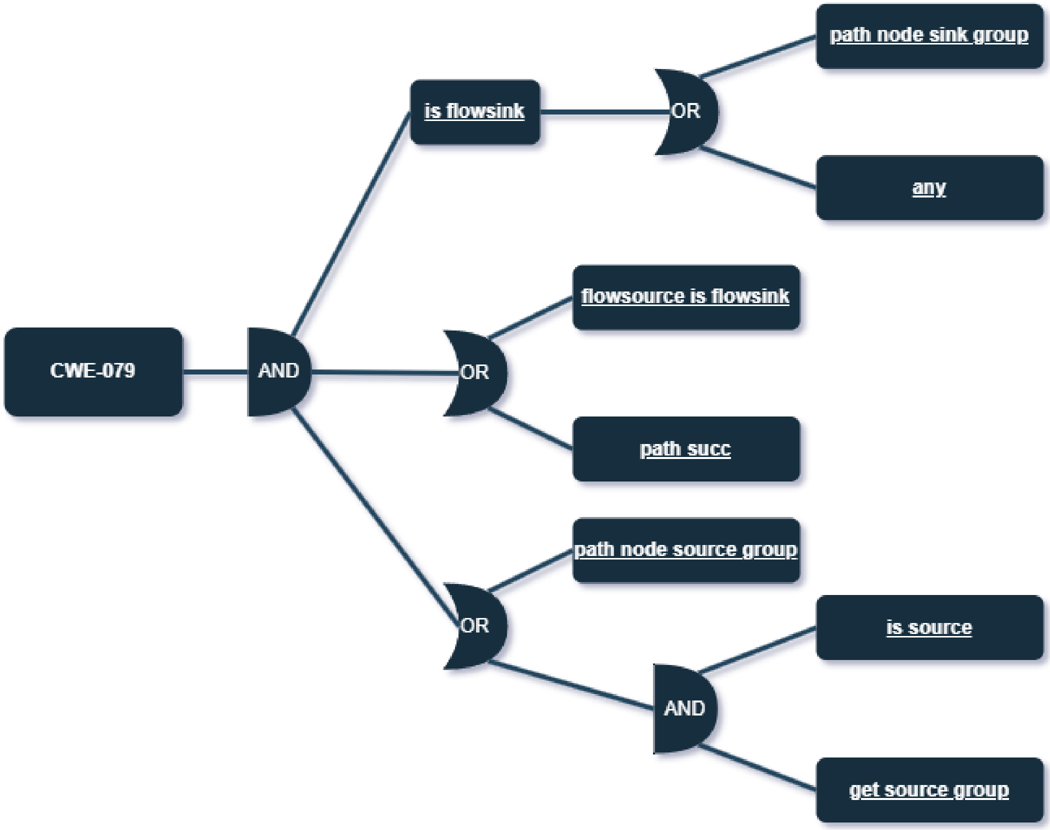

Figure 5.2: How subqueries are used to emulate CWE-079

5.3 Creating the fixed version of Vul4J-34

Since Vul4J provides human-made patches to its vulnerable programs, fixing Vul4-34 only required us to look through the mapping of each vulnerability to its repair patch, download said patch and apply it to the vulnerable code. In this case, the fix to Vul4J-34 is an update to core/src/main/-java/com/opensymphony/xwork2/interceptor/I18nInterceptor.java provided at `https://github.com/apache/struts/commit/fc2179cf1ac9fbfb61e3430fa88b641d87253327`.

To validate the program was now fixed, we ran Vul4J's integrated test suite that served as proof of vulnerability (PoV) and none of the tests was triggered. Since we didn't have any PoV, we assumed the program was rendered non-vulnerable.

5.4 Preparing the source code

Since PyGGi's line mutations don't discriminate between comment lines and code lines, we wrote a small script tasked with the removal of every line of comments from a file, eliminating the chances of having lines of comments mutated and ensuring line mutations will have an effect on the program by modifying lines of code.

This script only works for comments similar to Java's and works by rewriting target files with modified versions of the code where lines of comments are suppressed. No backups of the modified files are made. The script itself can be found in the rmvComments.py file. Its source code is also available, figure 5.3.

```
import pyparsing
from pathlib import Path

for file in Path("./VUL4J-34").rglob(’*’):
    if(file.name.endswith(".java")):
        with open(file, ’r’) as fileR:
            fileContent = fileR.read()
        with open(file, ’w’) as fileW:
            commentFilter = pyparsing.javaStyleComment.suppress()
            fileContent = commentFilter.transformString(fileContent)
            fileW.write(fileContent)

print("Done")
```

Figure 5.3: Script to remove comments from Java files

5.5 Running VulGr

The results of running VulGr are listed in table 5.3. According to these 10 tests, VulGr was able to inject a vulnerability into the repaired program in only one generation and as soon as a compilable mutated source code was generated. The results are discussed in 6.2.1 as they were negatively impacted by false positives.

VulGr Run	Result	Subqueries Execution Time	Database Creation Time	Last Generation Number	Fittest Individual Number for the Generation
1	Vulnerable	624.6s	492.1s	1	2
2	Vulnerable	235.7s	426.3s	1	3
3	Vulnerable	255.5s	548.5s	1	1
4	Vulnerable	209.2s	481.3s	1	2
5	Vulnerable	252.6s	475.4s	1	1
6	Vulnerable	209.2s	429.8s	1	1
7	Vulnerable	229.9s	494.4s	1	2
8	Vulnerable	228.8s	538.3s	1	2
9	Vulnerable	227s	518.9s	1	1
10	Vulnerable	213.9s	444.7s	1	1

Table 5.3: Results of the initial runs of VulGr

As mentioned previously, once a query has been compiled, it is cached by CodeQL. The beneficial effects of caching can be seen in the table, as the execution time of the subqueries for the first run took much longer than for the following ones.

5.6 Incorporating the Proof of Vulnerability

Since the previous results in table 5.3 were unreliable and caused the fitness function to falsely label non-vulnerable programs as vulnerable, prematurely stopping the run, we tried incorporating the proof of vulnerability provided with the Vul4J vulnerability. This was done in order to add a new layer of certitude when it came to assess if the subqueries detected a true or false positive.

The new fitness function was weighted equally between the original fitness function, making use of the subqueries, and Vul4J's proof of vulnerability, each accounting for 50 percent of the final fitness score.

The results of one run of VulGr are presented in tables A.1, A.2 and A.3 of the annex. One run is not sufficient to empirically give results as to the efficiency of this modified fitness function. Future works could explore this approach in more details, but since the values of the result can only go from 0.5 to 1 (CodeQL always returning a false positive, ensuring the minimal fitness function for a compilable variant is 0.5 and Vul4J-34's proof of vulnerability only able to return 0 or 0.5), this approach is unreliable and doesn't allow for gradual improvements over the generations in its current state.

Chapter 6

Interpretation

This chapter delves into our evaluation of the results and provides further explanations as to why the experiments yielded unsuccessful results.

6.1 Evaluation of the results

This process could not inject vulnerabilities into a repaired Vul4J program, the main reasons behind this unfortunate end to the experiment are that CodeQL's queries are not precise enough to correctly assess with certainty whether a program is vulnerable (section 6.2.1) and CodeQL's database creation and query execution are lengthy processes causing the Genetic Improvement framework using it to require an unrealistic amount of time to yield interesting results. This issue is explained in detail in section 6.2.2.

6.2 Negative factors

6.2.1 Accuracy of the results

As mentioned in 5.5, this run of PyGGi and VulGr did not manage to produce a vulnerable program. The reason behind this lack of result is the lack of accuracy of the queries, causing the results of CodeQL's subqueries to be inaccurate. We made this conclusion by observing that when our fitness function returned a fitness of 1 (completely fit individual which should be vulnerable according to the queries), the PoV test in Vul4J's test suite couldn't detect any vulnerability. The 4 percent success rate (1 out of 25) when using CodeQL's queries to detect vulnerabilities, as seen in table 5.2 and table 5.1, foreshadowed this issue.

This unfortunately means that CodeQL and the current security queries made by the community are not precise enough and suffer from the same issues as noted by Katerina Goseva-Popstojanova and Andrei Perhinsch in 2015 [9]. The biggest issues we discovered when running the subqueries through VulGr is that they are prone to find false positives, causing the fitness function to inaccurately believe that the program is vulnerable or closer to being vulnerable when it's not the case as the Vul4J test suite contradicts. Mutants have inflated fitness values, negatively impacting the overall process, which in turn did not contribute to reliably and automatically inject vulnerabilities into programs.

When augmenting the fitness function with Vul4J-34's proof of vulnerability, it quickly became apparent that the fitness score tended to be 0.5 when a compilable mutant was produced (50 percent vulnerable, due to CodeQL considering the mutated program as vulnerable and due to Vul4J's proof of vulnerability not detecting any vulnerability). When the code didn't compile, the fitness score naturally was 0. Since only the proof of vulnerability, consisting of only one test, could change the fitness score, it would theoretically be possible to generate a vulnerable individual with a fitness score of 1 though this would happen completely randomly and wouldn't benefit from the genetic improvement framework nor CodeQL. Mutations would occur randomly until the fitness score jumps from 0.5 to 1 when using CodeQL in the fitness function, or from 0 to 1 if we removed CodeQL from the fitness function.

6.2.2 Runtime

Getting results was time-consuming due to the lengthy process of rebuilding a database of each occurrence of mutated source code. In practical terms, it meant that for one generation of 10000 individuals, CodeQL had to build 10000 databases matching the 10000 mutated programs, each time from scratch since CodeQL doesn't currently support incremental updates of the database [18].

This is the reason individuals in A.1, A.2 and A.3 each took a few minutes to be evaluated by the fitness function, severely slowing down the process of applying GI to the repaired Vul4J-34 program. This time-consuming process effectively made it impossible to get any interesting results in a reasonable time frame given the hardware available for this experiment, namely Google's Colaboratory.

If we use the following formula for a single generation:

$$(N * cR) * (average(dT) + average(sT)$$

$$cR = 21/45$$

with N being the number of variants in a single generation, cR being the rate of programs that can actually be compiled (out of 45 variants, 21 could be compiled), *average* being a function returning the average value of the set of numbers given and dT and sT respectively being the Database creation Times and Subqueries execution Times of the results in A.1.

We get the following result:

$$(10000 * 0.466) * (481.1 + 162.2) = 2997778s$$

2997778 is equivalent to 832.7 hours or 34.69 days, more than a month.

Chapter 7

Conclusion

This chapter summarizes the processes that were applied, the results that were produced, our interpretation of the results. The answers to the research questions asked earlier in chapter 3 are also provided and ideas that emerged during the experiments as well as ideas for future works and improvements are suggested.

7.1 Summary

The aim of this experiment was to automatically inject security vulnerabilities into programs for the purpose of obtaining high-quality labelled vulnerable programs. These labelled programs could have then helped to solve the issue of data scarcity. They would have been provided as a new dataset for researchers training machine-learning models specialized in the detection of vulnerabilities and enabled them to follow the steps and logic applied in this experiment in order to generate their own vulnerable programs accommodating their specific needs.

The experiment was made using VulGr, a modified version of the PyGGi genetic improvement framework that was implemented in the context of this thesis. VulGr automatically applied line mutations to a non-vulnerable program, individuals generated through these mutations were then assessed by a fitness function making use of CodeQL, a static code analyser.

The experiment was unsuccessful, as the programs produced did not contain vulnerabilities, and the results of VulGr were invalidated by Vul4J's test suite. The most important threat to the validity of the results provided by VulGr was the inaccuracy of CodeQL's security queries, frequently unable to detect vulnerabilities in the first place or falsely considering non-vulnerable program as vulnerable (false positives). Furthermore, CodeQL's lengthy processes when analysing and looking for vulnerabilities in programs complicated the experiments and made it impossible for a GI framework to finish its run in an acceptable time frame in the context of this thesis.

7.2 Answers to the research questions

- **Question 1: How reliable is CodeQL compared to previously tested static code analysers when dealing with security vulnerabilities?**

  In its current state, CodeQL didn't perform adequately and is inaccurate most of the time when using its security queries on a database of a Vul4J program. It is however still in development and the security queries are an ongoing community effort and the accuracy of these should improve with time.

CodeQL only achieved a 4 percent success rate of finding true positives when applied to Vul4J's first 25t vulnerable programs having a CWE vulnerability. It can be noted that in [9], three static code analysers are used in conjunction against a different dataset, the Juliet test suite. We thus can't empirically say CodeQL was less accurate than the static code analysers in [9], since the three tools were used in conjunction against a dataset they had been trained on and none of them were able to detect more than 27 percent of the vulnerabilities.

- **Question 2: How can CodeQL's take on source code analysis through databases of code be realistically used for the purpose of GI?**

  At the moment, CodeQL's database and querying systems are too time-consuming to be realistically used in a GI process, where thousands of databases have to be built, and even more queries have to be run. When using Colaboratory, building a database of a mutated program and running queries against it took several minutes, repeating the same processes for thousands of individuals would take more than a month for a single generation at the current pace (as calculated in section 6.2.2).

  However, an update to allow databases to be incrementally built, greatly reducing the build time by updating the previous databases with the new modifications, is currently in the works and could significantly speed up the creation of the database [18]. Similarly, queries are continually improved and multiple subqueries can be run in parallel, granted a hardware able to offer powerful multi-threading capabilities. It can thus be expected that in the future, this constraint will disappear and allow for CodeQL's take on source code analysis to be realistically used for the purpose of GI, producing results within an acceptable time frame.

- **Question 3: How can genetic improvement be used in the field of automatic vulnerability injection?**

  Based on the answers of the two previous questions, it is currently not possible to use genetic improvement for the purpose of automatic vulnerability injection. The approach through GI requiring thousands of iterations, which each require a non-negligible time to be assessed, and using inaccurate tools could not automatically inject vulnerabilities into Vul4J-34.

  Similarly to the answers to the previous questions, updates and community efforts have the potential to allow for a practical application of GI to produce vulnerable programs following the process and steps detailed in 4.

7.3 Parallel and future works

7.3.1 Using Genetic Improvement to improve the accuracy of CodeQL's queries

Since CodeQL's security queries suffer from a lack of accuracy, genetic improvement could be applied to them in the aim of making them more accurate. A typical workflow for such an application of GI would have been to isolate a program with a vulnerability not detected by the corresponding security query and mutating said query through statement and grammar-based mutations. The mutated query would then be run against the vulnerable program to assess its accuracy and ability to detect a true positive vulnerability and avoid false positives and false negatives.

We didn't explore this idea thoroughly due to the theme of the thesis being focused around the generation of vulnerable programs and the lack of a reliable way to guide the mutations toward a higher accuracy. Indeed, the fitness function would only be able to assess whether the mutated query was able to find the vulnerability or not. It would be unable to detect if a mutated query is closer to getting a true positive than the original query.

7.3.2 Using better hardware and waiting for improvements to CodeQL

This experimentation could be revisited in the future. The time constraints that resulted in a lack of concrete results could be met through the use of specialized hardware, benefitting from better computing capabilities and more threads. Incremental building of CodeQL databases would also significantly speed up the time it takes to assess mutated programs [18], alleviating the issue. Using a smaller and less complex program in which to inject a vulnerability would also translate in a less complex CodeQL database, in turn speeding up the execution of queries against it.

7.3.3 Optimizing the genetic improvement process

Since the experiments did not yield valid results, it was hard to tailor VulGr to generate a vulnerable program more reliably or faster. However, if valid results were produced, the next steps would have been to try multiple mutation techniques, trying other approaches to selection and look for optimal values regarding the number of generations and individuals for each generation in the aim of tailoring VulGr to generate vulnerabilities more efficiently.

7.4 Contributions of the thesis

7.4.1 State of CodeQL's security capabilities

The experimentations conducted in this thesis were able to demonstrate CodeQL's security queries were unable to detect vulnerabilities in the vast majorities of the tests that were run. Out of 25 Vul4J programs, only one's vulnerability was detected by CodeQL's security queries, achieving a 4 percent success rate.

Using CodeQL for security purposes is currently not sufficient to detect vulnerabilities and shouldn't be used alone when developing CWE compliant software programs.

7.4.2 A framework for future successful injection of vulnerabilities through genetic improvement

This thesis describes an approach to using genetic improvement to inject vulnerabilities into non-vulnerable programs. Its overall workflow can help understand genetic improvement and be transposed to other uses. The approach can also be modified, for example future works may want to use a different oracle than CodeQL for the implementation of the fitness function.

The thesis also includes the code of VulGr and the Colaboratory used to conduct the experiments. VulGr being modular, updating CodeQL queries, modifying the fitness function, changing the mutation techniques and so on is easy and aims to facilitate future modifications.

Bibliography

[1] Hasanen Alyasiri. “Evolving Rules for Detecting Cross-Site Scripting Attacks Using Genetic Programming”. In: *Advances in Cyber Security: Second International Conference, ACeS 2020, Penang, Malaysia, December 8-9, 2020, Revised Selected Papers 2*. Springer. 2021, pp. 642–656.

[2] Gabin An et al. “PyGGI 2.0: Language Independent Genetic Improvement Framework”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: ACM, 2019, pp. 1100–1104. ISBN: 978-1-4503-5572-8. DOI: `10.1145/3338906.3341184`. URL: `http://doi.acm.org/10.1145/3338906.3341184`.

[3] Benjamin Aziz, Mohamed Bader, and Cerana Hippolyte. “Search-based sql injection attacks testing using genetic programming”. In: *Genetic Programming: 19th European Conference, EuroGP 2016, Porto, Portugal, March 30-April 1, 2016, Proceedings 19*. Springer. 2016, pp. 183–198.

[4] Paul E Black et al. “SARD: Thousands of reference programs for software assurance”. In: *J. Cyber Secur. Inf. Syst. Tools Test. Tech. Assur. Softw. Dod Softw. Assur. Community Pract* 2.5 (2017).

[5] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Dıaz Ferreyra. “Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques”. In: *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 2022, pp. 464–468. DOI: `10.1145/3524842.3528482`.

[6] Bénimédourène Charles. *PyGGi(VulGr) + CodeQL Colaboraty*. `https://colab.research.google.com/drive/1q7Y7lfUOG4pZvyL3I2x8pvuoOunmCkMU#scrollTo=fcYEo7ApuPvH`. 2023.

[7] Bénimédourène Charles. *VulGr*. `https://github.com/C-hrl/VulGr-PyGI`. 2023.

[8] Gabriel Díaz and Juan Ramón Bermejo. “Static analysis of source code security: Assessment of tools against SAMATE tests”. In: *Information and Software Technology* 55.8 (2013), pp. 1462–1476. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2013.02.005`. URL: `https://www.sciencedirect.com/science/article/pii/S0950584913000384`.

[9] Katerina Goseva-Popstojanova and Andrei Perhinschi. “On the capability of static code analysis to detect security vulnerabilities”. In: *Information and Software Technology* 68 (2015), pp. 18–33. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2015.08.002`. URL: `https://www.sciencedirect.com/science/article/pii/S0950584915001366`.

[10] William B Langdon and Mark Harman. “Optimizing existing software with genetic programming”. In: *IEEE Transactions on Evolutionary Computation* 19.1 (2014), pp. 118–135.

[11] Claire Le Goues. “Automatic program repair using genetic programming”. In: *Univ. Virginia, Charlottesville, VA, USA* (2013).

[12] Zhen Li et al. “A comparative study of deep learning-based vulnerability detection system”. In: *IEEE Access* 7 (2019), pp. 103184–103197.

[13] Elisa Boari de Lima et al. "Tuning Genetic Programming parameters with factorial designs". In: *IEEE Congress on Evolutionary Computation*. July 2010. DOI: 10.1109/CEC.2010.5586084.

[14] Aditya Nandy, Chenru Duan, and Heather J Kulik. "Audacity of huge: overcoming challenges of data scarcity and data quality for machine learning in computational materials discovery". In: *Current Opinion in Chemical Engineering* 36 (2022), p. 100778. ISSN: 2211-3398. DOI: https://doi.org/10.1016/j.coche.2021.100778. URL: https://www.sciencedirect.com/science/article/pii/S2211339821001106.

[15] Benjamin Petit. "Automatic vulnerability injection using natural language processing". 2022.

[16] Riccardo Poli, William Langdon, and Nicholas Mcphee. *A Field Guide to Genetic Programming*. Jan. 2008. ISBN: 978-1-4092-0073-4.

[17] Mohamed Shahin. "State-of-the-art review of some artificial intelligence applications in pile foundations". In: *Geoscience Frontiers* 130 (Nov. 2014). DOI: 10.1016/j.gsf.2014.10.002.

[18] Tamás Szabó and Max Schaefer. *Incremental CodeQL*. https://githubnext.com/projects/incremental-codeql/. 2023.

[19] Westley Weimer et al. "Automatically finding patches using genetic programming". In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 364–374.

Appendix A

Annex

A.1 VulGr operating 45 mutations

Individual	Database Creation Time	Subqueries Execution Time	Fitness Score
1	Compilation Failed	Skipped	0
2	Compilation Failed	Skipped	0
3	468s	232.9s	0.5
4	Compilation Failed	Skipped	0
5	466s	212.7s	0.5
6	497.7s	227.6s	0.5
7	Compilation Failed	Skipped	0
8	Compilation Failed	Skipped	0
9	Compilation Failed	Skipped	0
10	Compilation Failed	Skipped	0
11	480.1s	228.9s	0.5
12	511.5s	212.1s	0.5
13	Compilation Failed	Skipped	0
14	481.5s	217.7s	0.5
15	Compilation Failed	Skipped	0

Table A.1: Results 1-15 of running VulGr with the fitness function augmented with Vul4J's PoV

Individual	Database Creation Time	Subqueries Execution Time	Fitness Score
16	465s	222.6s	0.5
17	Compilation Failed	Skipped	0
18	Compilation Failed	Skipped	0
19	Compilation Failed	Skipped	0
20	474.4s	212.7s	0.5
21	Compilation Failed	Skipped	0
22	Compilation Failed	Skipped	0
23	458.8s	221.0s	0.5
24	475.6s	241.5s	0.5
25	Compilation Failed	Skipped	0
26	486.4s	208.6s	0.5
27	496.4s	244.5s	0.5
28	494.8s	223.6s	0.5
29	Compilation Failed	Skipped	0
30	474.5s	239.1s	0.5

Table A.2: Results 16-30 of running VulGr with the fitness function augmented wit Vul4J's PoV

Individual	Database Creation Time	Subqueries Execution Time	Fitness Score
31	484.3	211.3	0.5
32	490.4	237.5	0.5
33	Compilation Failed	Skipped	0
34	457.6	221.6	0.5
35	498.6	235.3	0.5
36	Compilation Failed	Skipped	0
37	Compilation Failed	Skipped	0
38	487.1	207.1	0.5
39	485.4	224.3	0.5
40	Compilation Failed	Skipped	0
41	468.9	237.2	0.5
42	Compilation Failed	Skipped	0
43	Compilation Failed	Skipped	0
44	Compilation Failed	Skipped	0
45	Compilation Failed	Skipped	0

Table A.3: Results 31-45 of running VulGr with the fitness function augmented wit Vul4J's PoV

A.2 Mapping of each Vul4J to its potential CWE vulnerability

VUL4J ID	CWE ID	CWE NAME
VUL4J-1	CWE-20	Improper Input Validation
VUL4J-2	CWE-611	Improper Restriction of XML External Entity Reference
VUL4J-3	Not Mapping	Not Mapping
VUL4J-4	Not Mapping	Not Mapping
VUL4J-5	Not Mapping	Not Mapping
VUL4J-6	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')
VUL4J-7	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')
VUL4J-8	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')
VUL4J-9	Not Mapping	Not Mapping
VUL4J-10	CWE-20	Improper Input Validation
VUL4J-11	CWE-264	Permissions, Privileges, and Access Controls
VUL4J-12	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')
VUL4J-13	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')
VUL4J-14	CWE-20	Improper Input Validation
VUL4J-15	CWE-611	Improper Restriction of XML External Entity Reference
VUL4J-16	CWE-264	Permissions, Privileges, and Access Controls
VUL4J-17	Not Mapping	Not Mapping
VUL4J-18	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
VUL4J-19	Not Mapping	Not Mapping
VUL4J-20	Not Mapping	Not Mapping
VUL4J-21	CWE-254	7PK - Security Features
VUL4J-22	CWE-284	Improper Access Control
VUL4J-23	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
VUL4J-24	CWE-611	Improper Restriction of XML External Entity Reference
VUL4J-25	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
VUL4J-26	CWE-20	Improper Input Validation
VUL4J-27	CWE-264	Permissions, Privileges, and Access Controls
VUL4J-28	CWE-264	Permissions, Privileges, and Access Controls
VUL4J-29	CWE-264	Permissions, Privileges, and Access Controls
VUL4J-30	CWE-20	Improper Input Validation
VUL4J-31	Not Mapping	Not Mapping
VUL4J-32	Not Mapping	Not Mapping
VUL4J-33	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')
VUL4J-34	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
VUL4J-35	CWE-352	Cross-Site Request Forgery (CSRF)
VUL4J-36	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')
VUL4J-37	CWE-502	Deserialization of Untrusted Data
VUL4J-38	CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')
VUL4J-39	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor
VUL4J-40	CWE-287	Improper Authentication

Table A.4: Vul4J-CWE mapping of vulnerabilities 1 to 40

VUL4J ID	CWE ID	CWE NAME
VUL4J-41	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
VUL4J-42	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
VUL4J-43	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
VUL4J-44	CWE-310	Cryptographic Issues
VUL4J-45	CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')
VUL4J-46	Not Mapping	Not Mapping
VUL4J-47	CWE-611	Improper Restriction of XML External Entity Reference
VUL4J-48	CWE-20	Improper Input Validation
VUL4J-49	CWE-20	Improper Input Validation
VUL4J-50	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
VUL4J-51	CWE-918	Server-Side Request Forgery (SSRF)
VUL4J-52	CWE-269	Improper Privilege Management
VUL4J-53	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')
VUL4J-54	CWE-502	Deserialization of Untrusted Data
VUL4J-55	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')
VUL4J-56	CWE-918	Server-Side Request Forgery (SSRF)
VUL4J-57	CWE-532	Insertion of Sensitive Information into Log File
VUL4J-58	CWE-863	Incorrect Authorization
VUL4J-59	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
VUL4J-60	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
VUL4J-61	CWE-611	Improper Restriction of XML External Entity Reference
VUL4J-62	CWE-287	Improper Authentication
VUL4J-63	Not Mapping	Not Mapping
VUL4J-64	CWE-611	Improper Restriction of XML External Entity Reference
VUL4J-65	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
VUL4J-66	CWE-20	Improper Input Validation
VUL4J-67	CWE-770	Allocation of Resources Without Limits or Throttling
VUL4J-68	CWE-20	Improper Input Validation
VUL4J-69	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
VUL4J-70	Not Mapping	Not Mapping
VUL4J-71	Not Mapping	Not Mapping
VUL4J-72	CWE-345	Insufficient Verification of Data Authenticity
VUL4J-73	CWE-522	Insufficiently Protected Credentials
VUL4J-74	CWE-332	Insufficient Entropy in PRNG
VUL4J-75	CWE-19	Data Processing Errors
VUL4J-76	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
VUL4J-77	CWE-502	Deserialization of Untrusted Data
VUL4J-78	CWE-502	Deserialization of Untrusted Data
VUL4J-79	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Table A.5: Vul4J-CWE mapping of vulnerabilities 41 to 79

A.3 Code of the 8 subqueries of CWE-079

```
/**
 * @name any
 * @description Writing user input directly to a web page
 *              allows for a cross-site scripting vulnerability.
 * @kind path-problem
 * @problem.severity error
 * @security-severity 6.1
 * @precision high
 * @id java/xss
 * @tags security
 *       external/cwe/cwe-079
 */

 import java
 import semmle.code.java.dataflow.FlowSources
 import semmle.code.java.security.XSS
 import DataFlow::PathGraph
 import semmle.code.java.dataflow.internal.DataFlowImpl

 class XssConfig extends TaintTracking::Configuration {
   XssConfig() { this = "XSSConfig" }

   override predicate isSource(DataFlow::Node source) { source instanceof
       RemoteFlowSource }

   override predicate isSink(DataFlow::Node sink) { sink instanceof XssSink }

   override predicate isSanitizer(DataFlow::Node node) { node instanceof
       XssSanitizer }

   override predicate isSanitizerOut(DataFlow::Node node) { node instanceof
       XssSinkBarrier }

   override predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::Node
       node2) {
     any(XssAdditionalTaintStep s).step(node1, node2)
   }
 }

 from DataFlow::PathNode source, DataFlow::PathNode sink, XssConfig conf
 where sink = any(PathNodeSink cSink | not exists(cSink.getSinkGroup())) and
     sink.getConfiguration() = conf
 select sink.getNode(), source, sink, "Cross-site scripting vulnerability due to a
     $@.", source.getNode(), "user-provided value"
```

Figure A.1: First subquery, nicknamed “Any”

```
/**
 * @name flowsource-is-flowsink
 * @description Writing user input directly to a web page
 *              allows for a cross-site scripting vulnerability.
 * @kind path-problem
 * @problem.severity error
 * @security-severity 6.1
 * @precision high
 * @id java/xss
 * @tags security
 *       external/cwe/cwe-079
 */

 import java
 import semmle.code.java.dataflow.FlowSources
 import semmle.code.java.security.XSS
 import DataFlow::PathGraph

 class XssConfig extends TaintTracking::Configuration {
   XssConfig() { this = "XSSConfig" }

   override predicate isSource(DataFlow::Node source) { source instanceof
       RemoteFlowSource }

   override predicate isSink(DataFlow::Node sink) { sink instanceof XssSink }

   override predicate isSanitizer(DataFlow::Node node) { node instanceof
       XssSanitizer }

   override predicate isSanitizerOut(DataFlow::Node node) { node instanceof
       XssSinkBarrier }

   override predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::Node
       node2) {
     any(XssAdditionalTaintStep s).step(node1, node2)
   }
 }

 from DataFlow::PathNode source, DataFlow::PathNode sink, XssConfig conf
 where source.getConfiguration() = conf and sink.getConfiguration() = conf and
     source = sink
 select sink.getNode(), source, sink, "Cross-site scripting vulnerability due to a
     $@.",
   source.getNode(), "user-provided value"
```

Figure A.2: Second subquery, nicknamed “Flowsource is Flowsink”

```
/**
 * @name is-flowsink
 * @description Writing user input directly to a web page
 *              allows for a cross-site scripting vulnerability.
 * @kind path-problem
 * @problem.severity error
 * @security-severity 6.1
 * @precision high
 * @id java/xss
 * @tags security
 *       external/cwe/cwe-079
 */

import java
import semmle.code.java.dataflow.FlowSources
import semmle.code.java.security.XSS
import DataFlow::PathGraph

class XssConfig extends TaintTracking::Configuration {
  XssConfig() { this = "XSSConfig" }

  override predicate isSource(DataFlow::Node source) { source instanceof
      RemoteFlowSource }

  override predicate isSink(DataFlow::Node sink) { sink instanceof XssSink }

  override predicate isSanitizer(DataFlow::Node node) { node instanceof
      XssSanitizer }

  override predicate isSanitizerOut(DataFlow::Node node) { node instanceof
      XssSinkBarrier }

  override predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::Node
      node2) {
    any(XssAdditionalTaintStep s).step(node1, node2)
  }
}

from DataFlow::PathNode source, DataFlow::PathNode sink, XssConfig conf
where conf.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "Cross-site scripting vulnerability due to a
    $@.",
  source.getNode(), "user-provided value"
```

Figure A.3: Second subquery, nicknamed “Is Flowsink”

```
/**
 * @name is-source
 * @description Writing user input directly to a web page
 *              allows for a cross-site scripting vulnerability.
 * @kind path-problem
 * @problem.severity error
 * @security-severity 6.1
 * @precision high
 * @id java/xss
 * @tags security
 *       external/cwe/cwe-079
 */

import java
import semmle.code.java.dataflow.FlowSources
import semmle.code.java.security.XSS
import DataFlow::PathGraph

class XssConfig extends TaintTracking::Configuration {
  XssConfig() { this = "XSSConfig" }

  override predicate isSource(DataFlow::Node source) { source instanceof
      RemoteFlowSource }

  override predicate isSink(DataFlow::Node sink) { sink instanceof XssSink }

  override predicate isSanitizer(DataFlow::Node node) { node instanceof
      XssSanitizer }

  override predicate isSanitizerOut(DataFlow::Node node) { node instanceof
      XssSinkBarrier }

  override predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::Node
      node2) {
    any(XssAdditionalTaintStep s).step(node1, node2)
  }
}

from DataFlow::PathNode source, DataFlow::PathNode sink, XssConfig conf
where source.getConfiguration() = conf and source.isSource()
select sink.getNode(), source, sink, "Cross-site scripting vulnerability due to a
    $@.",
source.getNode(), "user-provided value"
```

Figure A.4: Second subquery, nicknamed “Is Source”

```
/**
 * @name not-getsourcegroup
 * @description Writing user input directly to a web page
 *            allows for a cross-site scripting vulnerability.
 * @kind path-problem
 * @problem.severity error
 * @security-severity 6.1
 * @precision high
 * @id java/xss
 * @tags security
 *      external/cwe/cwe-079
 */

import java
import semmle.code.java.dataflow.FlowSources
import semmle.code.java.security.XSS
import DataFlow::PathGraph
import semmle.code.java.dataflow.internal.DataFlowImpl


class XssConfig extends TaintTracking::Configuration {
  XssConfig() { this = "XSSConfig" }

  override predicate isSource(DataFlow::Node source) { source instanceof
      RemoteFlowSource }

  override predicate isSink(DataFlow::Node sink) { sink instanceof XssSink }

  override predicate isSanitizer(DataFlow::Node node) { node instanceof
      XssSanitizer }

  override predicate isSanitizerOut(DataFlow::Node node) { node instanceof
      XssSinkBarrier }

  override predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::Node
      node2) {
    any(XssAdditionalTaintStep s).step(node1, node2)
  }
}

from DataFlow::PathNode source, DataFlow::PathNode sink, XssConfig conf
where source.getConfiguration() = conf and not
    exists(((PathNodeImpl)source).getSourceGroup())
select sink.getNode(), source, sink, "Cross-site scripting vulnerability due to a
    $@.",
source.getNode(), "user-provided value"
```

Figure A.5: Second subquery, nicknamed “Not Getsourcegroup”

```
/**
 * @name path-node-sink-group
 * @description Writing user input directly to a web page
 *              allows for a cross-site scripting vulnerability.
 * @kind path-problem
 * @problem.severity error
 * @security-severity 6.1
 * @precision high
 * @id java/xss
 * @tags security
 *       external/cwe/cwe-079
 */

import java
import semmle.code.java.dataflow.FlowSources
import semmle.code.java.security.XSS
import DataFlow::PathGraph
import semmle.code.java.dataflow.internal.DataFlowImpl


class XssConfig extends TaintTracking::Configuration {
  XssConfig() { this = "XSSConfig" }

  override predicate isSource(DataFlow::Node source) { source instanceof
      RemoteFlowSource }

  override predicate isSink(DataFlow::Node sink) { sink instanceof XssSink }

  override predicate isSanitizer(DataFlow::Node node) { node instanceof
      XssSanitizer }

  override predicate isSanitizerOut(DataFlow::Node node) { node instanceof
      XssSinkBarrier }

  override predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::Node
      node2) {
    any(XssAdditionalTaintStep s).step(node1, node2)
  }
}

from DataFlow::PathNode source, DataFlow::PathNode sink, XssConfig conf
where sink.getConfiguration() = conf and sink instanceof PathNodeSinkGroup
select sink.getNode(), source, sink, "Cross-site scripting vulnerability due to a
    $@.",
source.getNode(), "user-provided value"
```

Figure A.6: Second subquery, nicknamed “Path Node Sink Group”

```
/**
 * @name path-node-source-group
 * @description Writing user input directly to a web page
 *              allows for a cross-site scripting vulnerability.
 * @kind path-problem
 * @problem.severity error
 * @security-severity 6.1
 * @precision high
 * @id java/xss
 * @tags security
 *       external/cwe/cwe-079
 */

import java
import semmle.code.java.dataflow.FlowSources
import semmle.code.java.security.XSS
import DataFlow::PathGraph
import semmle.code.java.dataflow.internal.DataFlowImpl


class XssConfig extends TaintTracking::Configuration {
  XssConfig() { this = "XSSConfig" }

  override predicate isSource(DataFlow::Node source) { source instanceof
      RemoteFlowSource }

  override predicate isSink(DataFlow::Node sink) { sink instanceof XssSink }

  override predicate isSanitizer(DataFlow::Node node) { node instanceof
      XssSanitizer }

  override predicate isSanitizerOut(DataFlow::Node node) { node instanceof
      XssSinkBarrier }

  override predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::Node
      node2) {
    any(XssAdditionalTaintStep s).step(node1, node2)
  }
}

from DataFlow::PathNode source, DataFlow::PathNode sink, XssConfig conf
where source.getConfiguration() = conf and source instanceof PathNodeSourceGroup
select sink.getNode(), source, sink, "Cross-site scripting vulnerability due to a
    $@.",
source.getNode(), "user-provided value"
```

Figure A.7: Second subquery, nicknamed “Path Node Source Group”

```
/**
 * @name path-succ
 * @description Writing user input directly to a web page
 *              allows for a cross-site scripting vulnerability.
 * @kind path-problem
 * @problem.severity error
 * @security-severity 6.1
 * @precision high
 * @id java/xss
 * @tags security
 *       external/cwe/cwe-079
 */

 import java
 import semmle.code.java.dataflow.FlowSources
 import semmle.code.java.security.XSS
 import DataFlow::PathGraph

 import semmle.code.java.dataflow.internal.DataFlowImpl

 class XssConfig extends TaintTracking::Configuration {
   XssConfig() { this = "XSSConfig" }

   override predicate isSource(DataFlow::Node source) { source instanceof
       RemoteFlowSource }

   override predicate isSink(DataFlow::Node sink) { sink instanceof XssSink }

   override predicate isSanitizer(DataFlow::Node node) { node instanceof
       XssSanitizer }

   override predicate isSanitizerOut(DataFlow::Node node) { node instanceof
       XssSinkBarrier }

   override predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::Node
       node2) {
     any(XssAdditionalTaintStep s).step(node1, node2)
   }
 }

 from DataFlow::PathNode source, DataFlow::PathNode sink, XssConfig conf
 where pathSuccPlus(source, sink) and source.getConfiguration() = conf and
     sink.getConfiguration() = conf
 select sink.getNode(), source, sink, "Cross-site scripting vulnerability due to a
     $@.",
 source.getNode(), "user-provided value"
```

Figure A.8: Second subquery, nicknamed “Path Succ”