



Finding Security Issues in (Open Source) Software Repositories

Zer Jun Eng

supervised by

Dr. Achim BRUCKER

This report is submitted in partial fulfilment of the requirement for the
degree of MEng Software Engineering by Zer Jun Eng

COM3610

29th April 2019

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name : Zer Jun Eng

Date : 29th April 2019

Abstract

In Free/Libre and Open Source Software (FLOSS) repositories, some developers chose to not publicly disclose the details of various security vulnerability patches. Finding these vulnerability-fixing commits in a software repository could provide valuable insights for research purposes. This project aims to develop a repository mining tool that runs an analysis process to find vulnerability-fixing commits in Git repositories. The tool utilises regular expressions, static analysis techniques, and various approaches from software repository mining in the analysis process. The results collected provided considerable empirical evidence that shows the number of security issues fixed in 52 sample repositories from GitHub. After manual evaluation, it has shown that the tool has a true positive rate of 44.28%. Several problems and limitations of the developed tool are identified during the evaluation. Overall, the project has reached its goal, and it ends in the discussion of possible improvements to be made in the future.

Acknowledgements

I would like to thank my parents for their unconditional love and the full financial support throughout my university life. It would not be possible for me to finish this project and my course without them.

I would also like to thank my supervisor, Dr. Achim Brucker for continuously providing constructive advice for my project. I am honoured to work with you, and I look forward to working with you in the future.

Contents

1	Introduction	1
1.1	Background	1
1.2	Objectives	2
1.3	Challenges	3
1.4	Report Structure	4
1.5	Relationship to Degree Programme	4
2	Literature Review	5
2.1	Open Source Security	5
2.2	Taxonomy of Software Vulnerabilities	6
2.2.1	Common Weakness Enumeration	6
2.2.2	Common Vulnerabilities and Exposures	6
2.3	Security Issues in Open Source Software	7
2.3.1	Using Components with Known Vulnerabilities	7
2.4	Mining Software Repositories	9
2.4.1	Keywords Search	10
2.4.2	Properties of Vulnerability-Fixing Commits	10
2.4.3	Finding Security Vulnerabilities	11
2.4.4	Source Code Analysis	13
2.5	Improving Software Security with Static Analysis and Repository Mining	13
2.6	Usage of the Repository Mining Tool in Other Areas	14
2.6.1	Bugs Finding Model	14
2.6.2	Machine Learning Model for Automated Vulnerability Prediction	14

CONTENTS

3	Requirements and Analysis	16
3.1	Project Objectives	16
3.2	Software Requirements and Scope	17
3.2.1	Functional Requirements	17
3.2.2	Non-functional Requirements	18
3.2.3	Scope	18
3.3	Analysis	19
3.3.1	Programming Language	19
3.3.2	Libraries and Tools	19
3.3.3	File Format of Result	20
3.4	Proposed Method	20
3.5	Problems and Constraints	21
3.5.1	Quality of Commit Messages	21
3.5.2	Code Changes Analysis	22
3.5.3	Quality and Validity of Results	23
3.6	Testing	24
3.6.1	Unit Testing	24
3.6.2	System Testing	24
3.7	Evaluation	24
3.7.1	Real-world Projects Evaluation	25
3.7.2	Quality Evaluation	25
3.8	Ethical Issues	26
4	Design	27
4.1	Programming Practices	27
4.2	Implementation Model	28
4.2.1	Considered implementation models	28
4.2.2	Chosen implementation model	29
4.3	Supported Programming Languages	30
4.4	Overview of the Repository Mining Tool	31
4.4.1	Structure	31
4.4.2	Program Flow	32
4.5	Commit Messages Matching	33
4.5.1	Vulnerability Patterns and Regular Expressions	33

CONTENTS

4.5.2	Expected Behaviour	34
4.6	Vulnerable Code Searching	34
4.6.1	Code Searching Techniques	34
4.6.2	Chosen Technique	35
4.6.3	Severity and Confidence Level	36
4.7	Level of Detail in Result File	36
4.8	Statistical Evaluation	37
5	Implementation and Testing	38
5.1	Project Environment and Structure	38
5.1.1	Hardware Requirements	38
5.1.2	Software Requirements	38
5.1.3	Structure	39
5.2	Commit Messages Matching	39
5.2.1	Vulnerabilities	40
5.2.2	Writing the Regular Expressions	41
5.2.3	Searching for the Matches	42
5.3	Vulnerable Code Searching	43
5.3.1	Types of the Code Changes	43
5.3.2	Flawfinder and Bandit Integration	44
5.3.3	User-defined Programming Language	44
5.4	Configurable Options	45
5.5	Exceptions and Errors	46
5.5.1	Special Case	46
5.6	Testing	47
5.6.1	System Environment	47
5.6.2	Program Initialisation	47
5.6.3	Repository and Commit Properties	48
5.6.4	Commit Message Matching	48
5.6.5	Vulnerable Code Searching	49
5.6.6	Performance Testing	49
6	Results and Discussion	51
6.1	Data Set	51

CONTENTS

6.2	C/C++ Repositories	51
6.2.1	Case Study	52
6.3	Java Repositories	53
6.3.1	Case Study	53
6.4	Python Repositories	54
6.4.1	Case Study	54
6.5	Results Summary	55
6.6	Evaluation	57
6.6.1	Threats to Validity of Results	57
6.6.2	Problems and Limitations	58
6.6.3	True Positive Rate	59
6.7	Goals Achieved	60
6.7.1	Objectives	60
6.7.2	Functional Requirements	60
6.7.3	Non-functional Requirements	61
6.8	Further Work	61
7	Conclusion	62
	Acronyms	64
	Bibliography	65
	Appendices	78
A	Performance Testing Results	78
B	RegExp match, all repositories	80
C	RegExp match, C/C++ repositories	81
D	RegExp match, Java repositories	82
E	RegExp match, Python repositories	82

List of Figures

2.1	Examples of vulnerability-fixing commit.	11
3.1	A comparison between a CVE -identified fixing commit and a common vulnerability-fixing commit.	19
3.2	A comparison between a bad quality and a good quality commit message.	22
3.3	An example of code changes to prevent SQL injection [33]. . .	23
4.1	UML class diagram of Shefmine, the repository mining tool. .	31
4.2	UML activity diagram of the repository mining tool	32
4.3	An example of a search query ‘ CVE ’ on GitHub.	34
5.1	Directory struture of the project	39
5.2	An example of using capturing groups and alternation in regular expression.	41
5.3	Number of potentially vulnerable commits found since 1990 in all repositories analysed.	49
6.1	Number of potentially vulnerable commits found since 1990 in all repositories analysed.	56

List of Tables

3.1	Functional requirements of the mining tool.	18
3.2	Non-functional requirements of the mining tool.	18
3.3	Documentation format of the test case.	24
5.1	Test cases of system environment.	47
5.2	Test cases of program initialisation.	48
5.3	Test cases of repository and commit properties.	48
5.4	Test cases of commit message matching.	48
5.5	Test cases of vulnerable code searching.	49
6.1	Top 5 vulnerabilities matched by the regular expressions in the C/C++ repositories analysed.	52
6.2	Partial statistics of the Linux repository analysis.	52
6.3	Top 5 vulnerabilities matched by the regular expressions in the Java repositories analysed.	53
6.4	Partial statistics of the Tomcat repository analysis.	53
6.5	Top 5 vulnerabilities matched by the regular expressions in the Python repositories analysed.	54
6.6	Partial statistics of the Tomcat repository analysis.	55
6.7	Top 5 vulnerabilities matched by the regular expressions in all repositories analysed.	55
6.8	Achievement status of functional requirements in Table 3.1. .	60
6.9	Achievement status of non-functional requirements in Table 3.2.	61

LIST OF TABLES

A1	The time taken to complete the analysis process of each repository.	80
A2	All vulnerabilities matched by the regular expressions in all repositories analysed.	81
A3	All vulnerabilities matched by the regular expressions in the C/C++ repositories analysed.	81
A4	All vulnerabilities matched by the regular expressions in the Java repositories analysed.	82
A5	All vulnerabilities matched by the regular expressions in the Python repositories analysed.	83

Listings

5.1	Creating a new instance of <code>Vulnerability</code>	40
5.2	A partial snippet of the result file of CPython [80] repository.	42
5.3	An example snippet of the Java ruleset.	45

Chapter 1

Introduction

1.1 Background

Free/Libre and Open Source Software (FLOSS) is a type of software whose license allows the users to inspect, use, change and redistribute the software's source code [21]. Since the introduction of the version control system, many repository hosting sites such as SourceForge [93], Google Code [36], and GitHub [34] have been launched. As a result, the participation of global communities into **FLOSS** projects have started to grow and different contributions were made to improve the software quality, which included fixing software vulnerabilities [28].

Building secure software is expensive, difficult, and time-consuming. It is necessary to know when and how a security vulnerability is fixed throughout the software lifecycle. Software components such as plugins and **application processing interfaces (APIs)** are usually developed by third-party developers and widely reused in both open source and closed source software [45]. An important factor of software security is determined by the information provided by the vendor of the software components for deciding whether to perform the security update. Hence, the users of software components are advised to check the **National Vulnerability Database (NVD)** [64] regularly for detailed information of the vulnerabilities identified in the software components used. Furthermore, it would be more helpful if the developers of

the software components recorded the list of changes or provide informative Git commit messages for every version update of their component.

To perform a risk assessment of a potentially vulnerable component, it is required to have a deep understanding of the vulnerability entry points. Yet, not all projects follow the **Common Vulnerabilities and Exposures (CVE)** format or publish **CVE**, and **CVE** reports are usually lack of technical details that attribute the specific entry points of the vulnerability, which is an important aspect in part of the risk assessment. By identifying the vulnerability-fixing commits, the vulnerable lines of code can be located, which allows the users to check if a vulnerable component is being used or not. However, some developers believe that public disclosure of security vulnerabilities patch is dangerous, thus vulnerability-fixing commits are not commonly identified and recorded specifically in some open source software repositories to prevent malicious exploits [9]. As a result, there is a practical difficulty in applying this analysis approach to find the security relevant commits that are not documented using **CVE** or a similar format, which are known as the silent patches.

To address these issues, a repository mining tool that investigates commit messages and identifies vulnerable software components can be developed to reduce the time and cost required to mitigate the vulnerabilities. The repository mining tool should be able to detect the silent patches through an advanced process, which the tool must analyse the source code changes between commits to locate the vulnerable lines of code. Moreover, the mining tool should be applicable to all types of software projects that are using Git as their version control system. Projects that are using a different version control system are also supported after they have been migrated to Git.

1.2 Objectives

- Identify the security patterns of the security issues in the **Open Web Application Security Project (OWASP)** Top Ten Project. The tool is aimed to cover the most common or important security issues in the list.

- Develop a repository mining tool to search through the commit history of a repository and find a list of commit messages that match the patterns. The list should be produced in **JavaScript Object Notation (JSON)** file format.
- Extend the mining tool which checks the code difference in the commits found to obtain the actual commits fixing the security vulnerabilities.
- Create a statistical tool that reads the output file and reports a detailed analysis of the results.

1.3 Challenges

This section is a brief summary of the main challenges that might occurred during the project. A more thorough analysis of the problems and constraints is carried out in **Section 3.5**.

- **Data:** There are a large number of open source repositories available on GitHub. However, it is challenging to find a set of sample repositories that can produce accurate and consistent results.
- **Misclassification:** The commit messages for the same vulnerability patch are not always the same, thus misclassification is inevitable. Using regular expressions to match the patterns in the mining process do not guarantee the correctness of the result.
- **Evaluation:** After mining a list of commits that contain the identified patterns in its message, the evaluation process might not correctly locate the lines of code that addressed the security vulnerability. It might be required to perform a manual evaluation to correctly identify some of the results.
- **Time:** Large repository such as Linux which has more than 820,000 commits in total [49] could be extremely time-consuming for the repository mining tool to complete the search and evaluation process.

1.4 Report Structure

Chapter 2 reviews a range of academic articles, theories, and previous studies that is related to this project, as well as investigating the techniques and tools to be used.

Chapter 3 is a list of detailed requirements and a thorough analysis of design, implementation and testing stage. Some core decisions are reviewed in the analysis part to ensure the feasibility of the project.

Chapter 4 is a comparison between different design concepts, where the advantages and disadvantages of different approaches are stated. The chosen design is justified with suitable diagrams provided including wireframes and UML component diagrams.

Chapter 5 describes the implementation by highlighting novel aspects to the algorithms used. Testing is performed by following a suitable model to evaluate the implementation.

Chapter 6 presents all the results along with critical discussions about the main findings, and outlines the possible improvements that could be made in the future work.

Chapter 7 summarises the main points of previous chapters and emphasise the results found.

1.5 Relationship to Degree Programme

This project focuses on the research of real-world software security problems and offers valuable insights into computer security. By studying the patterns of security vulnerabilities patch in open source repositories, the practical knowledge for building and ensuring a secure system could be gained. Moreover, the difficulty of improving software security could be experienced during the evaluation process in this project. This relates to the Software Engineering degree as it requires a good understanding in version control system and it aims to improve software quality by reducing the time and effort needed to find security vulnerabilities in the source code.

Chapter 2

Literature Review

This chapter will start with the background contents of the project, and then focus on discussing the security aspect of open source software. Additionally, previous and existing relevant works are reviewed and a critical analysis is provided for the comparison of these resources and this project.

2.1 Open Source Security

The security of open source software mostly rely on the collaboration of the community. It is deduced that the power of open data and crowdsourcing will make open source security more reliable [39, 103], and provides more flexibility and freedom over the security option to their users [73]. However, when it comes to publishing the vulnerability information, it is suggested that the list of unconfirmed vulnerabilities should not be published publicly to protect the users from potential harms [89].

Arora, Nandkumar and Telang [37] have shown that vulnerabilities that are either secret or published but not patched attract fewer attacks than patched vulnerabilities. Although the research was conducted in 2006 and the results might be outdated, it still implies that developers might include a silent patch into some of the commits that is not explicitly recorded in the commit messages. It might be a rational approach for not disclosing the work attempted to fix a vulnerability, but other developers might not

be informed of the content change. Furthermore, if a similar vulnerability is discovered in the future, developers would need more effort for finding the previous solution. Therefore, it would be very useful for the developers if the mining tool developed in this project could detect the silent patches.

2.2 Taxonomy of Software Vulnerabilities

There are many software vulnerabilities being identified each year. By using a common vulnerability identifier system, vulnerability data can be shared across separate vulnerability databases to facilitate the interoperability of different tools. As this project focuses on finding security issues in open source repositories, it is necessary to discuss the industry-endorsed standard of software vulnerabilities categorisation.

2.2.1 Common Weakness Enumeration

The **Common Weakness Enumeration (CWE)** is a project launched by the Mitre Corporation and sponsored by the National Cyber Security Division of the United States Department of Homeland Security [23]. The **CWE** project organises the software weaknesses into a list of different categories, known as the **CWE** list. Software weaknesses are defined as errors that can lead to software vulnerabilities, which includes buffer overflows, authentication errors, code injection, etc. [24]. The **CWE** is now a formal standard for representing software weaknesses. Each entry in the **CWE** list contains detailed information about the specific weakness and is identified by a unique ID number.

2.2.2 Common Vulnerabilities and Exposures

The **Common Vulnerabilities and Exposures (CVE)** is another security project launched by the Mitre Corporation [25] to provide the community with a complete list of publicly known security vulnerabilities, known as the **CVE** entries. Each **CVE** entry is defined by an ID number, and includes a description followed by any relevant resources about the vulnerability. It

is now the standardised solution and industry-recognised standard for identifying vulnerabilities and exposures. However, developers and vendors are not required to publish security vulnerabilities of their projects **CVE** format. They are allowed to use their own naming scheme for the vulnerabilities, even if the same vulnerability has already been recorded in the **CVE** list.

2.3 Security Issues in Open Source Software

The **Open Web Application Security Project (OWASP)** is a worldwide non-profit organization committed to improve and raise the awareness of software security [70]. The project members of **OWASP** have worked together to produce a list of the most critical web application security risks based on the community feedback and comprehensive data contributed by different organizations. The list consists of ten categories of security attacks which are considered to be the most dangerous and popular in recent years. In **OWASP Top Ten 2017** [72], one of the vulnerabilities that is closely related to this project is ‘Using Components with Known Vulnerabilities’, which will be extensively discussed.

2.3.1 Using Components with Known Vulnerabilities

It has been indicated that a small software component could create a large error in a software system [12, 60, 90]. Components such as plugins, libraries, and modules are ubiquitous in both open source and proprietary software. Third-party components are increasingly being integrated into software to reduce the amount of time and effort required for development [10], but they also increase the risk of vulnerabilities being introduced into the software. These components are mostly maintained by different developers or organisations, and the time required to fix a vulnerability varies between developers. While the majority of third-party components are still being actively maintained after a long time, some of them might have depreciated and security patches are no longer being released. The users might continue to use a depreciated component if they could not find a better alternative. However, using outdated components greatly increase the risk of software exploits.

Therefore, for any large-scale system, the developers must scan for vulnerabilities regularly and subscribe to the security news related to the components used to reduce the risk of security vulnerabilities being introduced into the system.

Vulnerable components can be found using methods ranging from dependency checking to machine learning. While this project is related to the former approach, the latter approach concentrates on finding the relationship between software errors and vulnerabilities to identify or predict high-risk components. Dependency checking is an approach of detecting dependencies (plugins, libraries, etc.) with known vulnerabilities in a software. Several open source tools including **OWASP** Dependency Check [69], Retire.js [84], and Safety [87] are applications that identifies vulnerable dependencies in a software project. Cadariu et al. [16] have used the **OWASP** Dependency Check tool to find all known vulnerabilities that have a unique **CVE** identifier in proprietary software written in Java. According to their study, the **OWASP** Dependency Check tool has low precision due to the high false positives rate in the large data sets. However, Cadariu et al. [16] justified that the tool is still usable by taking into account that the checking process is automated and any security issue found is considered a valuable information for the users.

Machine learning-based approaches are also applicable to find vulnerable components in a software system. Briand, Basili and Hetmanski [15] have developed a model with Optimised Set Reduction (OSR) algorithm that uses set theory, predicate logic, probability, and vector in the calculation. The model focused on identifying the components that are more likely to produce a large number of errors and it was proved to be effective, but the main drawbacks are the complexity of the implementation and the extensive calculations required. In comparison to Briand’s approach, Scandariato et al. [88] have built a model that uses text mining techniques to predict vulnerable components. While Briand’s model is capable of identifying high-risk components, Scandariato’s model is able to predict vulnerabilities in the future releases of a software components, and the results achieved are satisfactory.

As a conclusion, static dependency checking tools provide a fast and easy

way to scan for vulnerable components, but the users are required to verify the validity and compatibility of the results with their software. In contrast, models that use machine learning technique has been proved to be effective and are more likely to produce consistent and accurate results. However, such models require a large amount of training data and are only designed for a specific area. While dependency checking approach is more related to the scope of this project, the capability of predicting vulnerable software components through machine learning is a great way of preventing severe software errors. In future work, machine learning could be incorporated into the tool developed in this project to improve its overall effectiveness.

2.4 Mining Software Repositories

Mining Software Repositories (MSR) is a process of collecting and analysing data from repositories, which includes version control repositories, mailing list repositories, and bug tracking repositories. **MSR** applies to a wide range of fields such as business, research, and security [77]. The purpose of **MSR** is to extract practical information from rich metadata and discover hidden trends about a specific evolutionary characteristic [43]. The information collected could be used in various development process. For example, some developers could gain insight by mining repositories, which may help them to enhance their software quality based on previous implementation evidence of other developers [38]. While **MSR** have various usages in different areas, the primary objective of this project will be focusing on finding the security issues in open source software repositories through **MSR**.

In order to identify both hidden and publicly disclosed patches, it is required to make effective use of **MSR** technique. A **MSR** process is normally carried out using tools or scripts made by the researchers themselves. Although there are many types of research in the **MSR** field in recent years, the majority of the tools or scrips used are not published publicly [85]. As a result, it is not possible to fully replicate the previous research methods and make improvements based on that. Despite the undisclosed information of research methods in many papers, Shang [91] suggested that the **MSR** pro-

cess should be split into several stages, with each stage focusing on a specific topic of the problem to achieve the optimal efficiency.

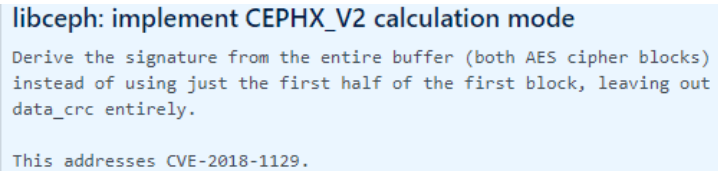
2.4.1 Keywords Search

For many complex approaches, the keywords searching process is considered to be the fundamental step. If the initial results produced in the searching stage is good, a huge amount of effort could be reduced in the later stages. However, the prerequisite is that the repository must have a sufficient amount of valuable information, which can be estimated by judging the history of the repository. To correctly and precisely retrieve the information for a query, it is required to integrate some algorithms and modules into the search function. Matsushita, Sasaki, and Inoue [55] developed a repository search system that makes use of two functions: lexical analysis function and token comparing function. The system produced very detailed results by deploying recursive search strategy using the keywords found into every commit. On the contrary, Mockus and Votta [57] designed an automated program that makes use of normalisation, word frequency analysis, and keyword clustering techniques to search the commit messages. Although the program is able to retrieve the results that include the keywords, the algorithm is unable to identify similar terms or inconsistent form of wording for the commit messages.

2.4.2 Properties of Vulnerability-Fixing Commits


For every vulnerability identified in a repository, the vulnerability-fixing process that involves analysis, implementation, testing, and release will be executed [68]. Most of the vulnerability-fixing commits are pushed during the implementation and testing stage of the process. However, if the commit message of a fixing commit is ambiguous, it will be challenging for any tools to determine the correctness of the commit. In order to analyse the common features of the vulnerability-fixing commits, it is needed to find these commits in open source repositories first. By analysing the properties of vulnerability-fixing commits, it will ease the implementation of the repository mining tool and enhance the quality of the regular expressions used.

Meneely et al. [56] conducted a research to study the properties of commits that introduce vulnerabilities, in which a reverse approach was used to find vulnerability-contributing commit by backtracking from the vulnerability-fixing commits. Meneely et al. identified the vulnerability-fixing commits by investigating into each vulnerability manually to find the respective fixing commit. However, the process of identifying vulnerability-fixing commit was not clearly explained by them, thus no constructive information about the properties of vulnerability-fixing commits is provided. On the contrary, Vásquez, Bavota and Velásquez [47] discovered that some vulnerability-fixing commits are grouped with other commits to address a vulnerability, which matches the assumption of Neuhaus and Plattner [61], and thus only part of the code committed is related to the vulnerability fix.



libceph: implement CEPHX_V2 calculation mode
 Derive the signature from the entire buffer (both AES cipher blocks) instead of using just the first half of the first block, leaving out data_crc entirely.
 This addresses CVE-2018-1129.

(a) Vulnerability fix in Linux kernel repository [48]



Merge pull request #360 from jehiah/csrf_validation_360
 CSRF protection for OAuth flow.

(b) Vulnerability fix in oauth2 proxy repository [65]

Figure 2.1: A comparison of (a) a higher quality and (b) a lower quality vulnerability-fixing commit.

2.4.3 Finding Security Vulnerabilities

It has been reported that the descriptions and references in vulnerability databases are often lack of complete documentation [53], and vulnerability-fixing commit are not ubiquitous in every open source repository [99]. Finding a security vulnerability could be hard if the resources available are limited. Having completed the researches on keywords searching techniques and prop-

erties of vulnerability-fixing commit, the approach for finding vulnerabilities can now be reviewed.

In this project, a static repository mining tool will be developed to find security vulnerabilities in open source repositories. Previous researches included the use of static software auditing and vulnerability mitigation tools to find bugs and vulnerabilities [20]. However, Bessey et al. [13] claimed that static tools have a negative effect on technical development due to its high false positives rate. While this statement might be true, it does not imply that all static tools are not effective as they differ in the techniques used in finding vulnerabilities [58]. Static tools usually require many experiments with different configurations to obtain the best result, and the result may vary across different data sets. This project differs from previous researches by aiming to find the security vulnerabilities through Git commit messages first, and then analyse the code changes in the commits. Researches have shown that vulnerability-fixing commits could be retrieved by extracting commit hashes from **CVE** references and gathering all commits that refer to a **CVE** number in its commit message [40], or by performing syntactic and semantic analysis on the commit messages [92]. To verify the validity of the commits retrieved, a screening test [29] can be performed to investigate the code changes in a commit against several criteria and identify the correct vulnerability-fixing commits.

This project extends prior work on Reis and Abreu's Secbench Mining Tool [82]. The tool aims to find vulnerabilities patch in GitHub repositories by using specific regular expressions for each vulnerability pattern. Then it creates a test case for every vulnerability found and these test cases are evaluated manually. Reis and Abreu [83] discussed the procedure of the evaluation and explained that human errors could occur due to source code complexity and similarity of vulnerability pattern. The approach of Secbench Mining Tool is similar to the concept of this project. However, performing manual evaluation on every result is not practical and it is proven that the use of automated algorithms can improve the detection process [51]. In this project, the tool developed should be able to automate the evaluation process to some extent, while preserving the accuracy of the results.

2.4.4 Source Code Analysis

This section is an extension to **Section 2.4.3**. As this project involves in analysing the code changes in Git commits using static analysis methods, it is necessary to review the techniques used to identify vulnerabilities by source code analysis tools.

Finding vulnerabilities by source code analysis technique is relatively difficult than analysing commit messages as it requires a high-level understanding in both software vulnerabilities and the programming language of the source code. Source code analysis tools are generally designed for a specific task, and only support specific programming languages [3]. There are two types of analysis methods: static and dynamic. This project will mainly focus on studying static analysis, and comparing its advantages and disadvantages with dynamic analysis.

One of the advantages of static source code analysis tool is that it can analyse the code without executing it [50], but this could also be the drawback as it might generate more false positives than dynamic analysis. Static analysis is considered to be a promising method for detecting possible and obvious security vulnerabilities [30]. Zitser, Lippmann, and Leek [104] have developed a static code analysis tool to find buffer overflow vulnerability in C programming language code. Their approach requires manual definitions of the overflow patterns in their tool, which is considered to be a general method in static analysis.

2.5 Improving Software Security with Static Analysis and Repository Mining

The methods discussed in **Section 2.4** could be used in conjunction with static analysis tool to improve software security. As common static source code analysis tools can only take one version of source code as input, it is unable to find security vulnerabilities in previous version of the source code unless it is specifically provided to the tool. Therefore, static analysis of source code could be integrated with **MSR** to find potential security vulner-

abilities in older versions of the source code.

Researches have suggested that users often turn off auto-updates for software [32, 54] to avoid possible consequences such as major interface changes or compatibility issues. This statement is justifiable if and only if an update does not introduce security improvements bug fixes. As a result, it is helpful to run static analysis in an older version of software and inform the users about the potential security vulnerabilities if the users decided not to update the software. This can be done by traversing through the history of a given software repository and reports the issues found in each revision, the implementation details are further explained in **Chapter 5**.

2.6 Usage of the Repository Mining Tool in Other Areas

While the repository mining tool developed in this project is only capable of finding security issues through Git commits, it can be extended or modified to aid the researches in other areas. Some examples are briefly discussed in the subsections below.

2.6.1 Bugs Finding Model

Both Williams and Hollingsworth [102] and Ostrand and Weyuker [67] utilised **MSR** technique in their bugs finding model. Williams and Hollingsworth mined the code changes in each commit to find possible bugs, while Ostrand and Weyuker mined the most frequently modified files between version releases to predict the bugs-prone files.

2.6.2 Machine Learning Model for Automated Vulnerability Prediction

In comparison to the static approach used in the mining tool, machine learning technique could be introduced to achieve higher reliability and accuracy on the result. Nguyen and Tran [62] and Perl et al. [76] have built their ma-

chine learning model with dependency graph and vulnerability-contributing commit as their main approach respectively, while Li et al. [46] and Russell et al. [86] used source code analysis method in their machine learning model. According to their researches, machine learning models are able to produce relatively high accuracy results.

Chapter 3

Requirements and Analysis

The purpose of this chapter is to express the aims in details and discuss the problems to be solved. This chapter will outline the requirements of the project and list the criteria to be met. The analysis part will cover every aspect of the design, implementation, and testing stage to ensure that the project is feasible.

3.1 Project Objectives

Initially, the objectives set in **Section 1.2** are an ideal concept of this project. Having completed the background research and literature review, it is now possible to provide a detailed description and more clearly defined objectives that improve the feasibility of this project.

1. **Vulnerability patterns:** The term ‘vulnerability pattern’ is used to represent the commit message pattern of different vulnerabilities. Correctly identifying the regular expression of each vulnerability pattern is time-consuming, and it would also need considerable refinement throughout the whole project. Hence, it might be more appropriate to reuse and improve the patterns provided in previous related works.
2. **Mining the commits:** This task involves creating a repository mining tool that makes extensive use of the pre-defined regular expressions to

search for the relevant commits. It is necessary to consider how closely a commit needs to match with the patterns for it to be included in the result. The file format for storing the results is **JSON**, and reasons are justified in **Section 3.3.3**.

3. **Evaluating the mined commits:** The mining tool can be extended to include a separate function that evaluates the commits mined to find the actual code commit addressing the security vulnerabilities. This project will consider to automate the evaluation process to some extent while maintaining the accuracy of the results at the standard level.

3.2 Software Requirements and Scope

3.2.1 Functional Requirements

Criteria	Importance
Compatibility: The mining tool should be able to run on all machines that meet the system requirements.	Essential
Completeness: The mining tool should be able to find all relevant commits of security vulnerabilities based on the regular expressions.	Essential
Informative: The statistical tool should return a complete analysis of the results.	Essential
Repeatable: The results should be repeatable and reproducible.	Essential
Scalability: The mining tool should be able to work on different project sizes, provided that the repository contains a certain amount of information.	Essential
Automated Evaluation: The process of classifying and evaluating the commits into different vulnerabilities patch should be automated to a certain extent.	Desirable
Extensibility: New vulnerability patterns and programming languages should be easily added into the tool.	Desirable

Robustness: The mining tool should be able to handle all possible errors without terminating the mining process.	Desirable
---	-----------

Table 3.1: Functional requirements of the mining tool.

3.2.2 Non-functional Requirements

Criteria	Importance
Code Style: The source code should be well-commented and follow a consistent coding style.	Essential
Documentation: Installation and user manual should be provided.	Essential
Lightweight: The mining tool should have minimal dependencies.	Essential
Open source: As the mining tool is built for researching open source repositories, it should be open source to suit the use cases.	Essential
Performance: The performance of the mining tool should be optimised for different project sizes.	Desirable

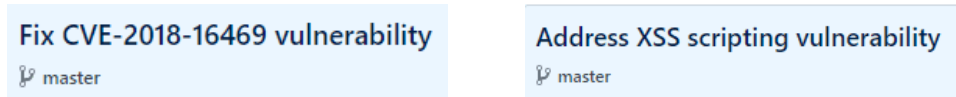
Table 3.2: Non-functional requirements of the mining tool.

3.2.3 Scope

In addition to **Table 3.1** and **Table 3.2**, several fundamental requirements can be included to define the scope of the mining tool.

- **Language in Commit Messages:** The mining tool will only search for commit messages that are written in English.
- **Programming Language:** The source code analysis function in the mining tool will support C, C++, Python 3, and Java.
- **Repository type:** The mining tool will only support Git repositories. Hence, repositories using other version control systems such will have to convert to Git first before using the mining tool.

- **Vulnerability type:** It is aimed that the mining tool should be able to detect **CVE**-identified vulnerabilities that have been fixed and recorded in the commit message **Figure 3.1 (a)**, and vulnerabilities that are not identified in **CVE** but recorded in the commit messages **Figure 3.1 (b)**.



(a) **CVE**-identified vulnerability fix [27] (b) Common vulnerability fix [19]

Figure 3.1: An example of (a) a **CVE**-identified vulnerability fix and (b) a common vulnerability fix.

3.3 Analysis

The aim of this section is to contemplate the options available for this project and review some of the fundamental decisions to be made before the implementation.

3.3.1 Programming Language

Python 3 [100] is chosen to be the main programming language for the repository mining tool. While other programming languages may be more suitable for tackling specific problems of this project, Python 3 provides sufficient coverage over every aspect with its comprehensive functionality. The greatest advantage of Python 3 is that it has a wide range of libraries that facilitate the development environment, which fully justified that a complete working solution can be produced using Python 3.

3.3.2 Libraries and Tools

The Python 3 standard library contains a wide range of built-in modules and extensive facilities. Moreover, many community created tools can be integrated into Python programs with minimal configurations as well. Below is a list of libraries and tools that will be used in the mining tool:

- The `re` [81] library for regular expression operations.
- The `json` [42] library for creating and reading the result file.
- GitPython [35] is a Python library built to simplify the interaction with Git repositories.
- PyDriller [94, 95] is a Python library built on top of GitPython with additional features for repository mining.
- Flawfinder [101] is a program written in Python that is designed to find potential security vulnerabilities in C/C++ source code.
- Bandit [11] is a tool created by the Python Code Quality Authority to find security issues in Python source code.
- `graudit` [52] is a simple source code auditing tool that finds potential security vulnerabilities in source code using regular expression searches.

3.3.3 File Format of Result

The **JavaScript Object Notation (JSON)** [41] has been chosen as the file format for storing the results in this project. This is because **JSON** is supported in Python and it does not require complicated operations in Python to access the data. While various alternative data interchange formats such as the **Extensible Markup Language (XML)** [31] has its unique advantages, it is important to choose a data interchange format that consumes less resource and have lower processing time for a large amount of data. Since it has been proved that **JSON** has better performance than **XML** in terms of processing time and resource utilisation [63], it is considered that **JSON** would be the best option for this project.

3.4 Proposed Method

This project strongly emphasises the need for finding security issues in open source repositories by mining software repositories. While it might be impossible to discover the security patches in a repository through a single search, the problem could be solved using divide and conquer. The ideal

concept of this project is to build a command-line interface program that is able to run two separate processes:

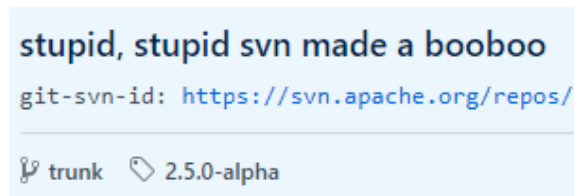
1. **Analysis:** This process takes a Git repository as input, then searches through the commit history and stores the list of potential vulnerability-fixing commits in a **JSON** file.
2. **Evaluation:** The evaluation process takes a **JSON** file as input, and present the results in an information way.

3.5 Problems and Constraints

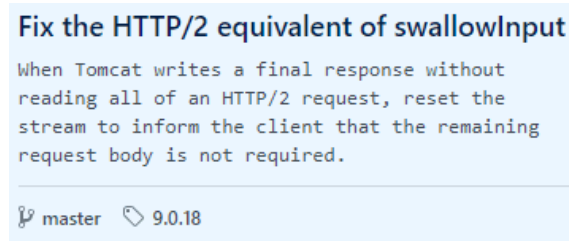
As mentioned in **Section 1.3**, the main challenges of this project are **data**, **misclassification**, **evaluation** and **time**. This section will discuss the problems in detail and review several ways of mitigating them, as well as analysing the possible constraints that might limit the achievements of the project.

3.5.1 Quality of Commit Messages

Although there are a lot of open source repositories available online, the majority of them does not have a formal guideline for the documenting the changes in the commit messages [14]. As part of the **data** problem, the commit messages in the real-world repositories (**Section 3.7.1**) might have lower quality compared to a self-created repository. It has been reported that the terms ‘fix’, ‘add’, and ‘test’ have the top average term frequency in the commit messages [1]. With these indistinct terms being widely used in the commit messages, the performance of the tool may drop on real-world repository test sets and it would require extra effort for finding the relevant vulnerability-fixing commits.



(a) Bad commit message quality [5]



```

Fix the HTTP/2 equivalent of swallowInput

When Tomcat writes a final response without
reading all of an HTTP/2 request, reset the
stream to inform the client that the remaining
request body is not required.

master 9.0.18

```

(b) Good commit message quality [6]

Figure 3.2: An example of (a) a bad quality commit message and (b) a good quality commit message.

Figure 3.2 shows a comparison between a bad quality and a good quality commit message. It should be noted that the commit message quality vary across different repositories and different authors. In real-world scenario, repositories that belong to organisations generally have a set of guidelines for documenting commit message. Such repositories would normally have higher quality commit message and are more likely to include valuable information in the commit messages too.

3.5.2 Code Changes Analysis

As discussed in **Section 3.5.1**, the commit messages in real-world repositories will contain noise and inconsistency that might affect the results retrieved. Therefore, it is expected that the results will contain a certain amount of false positive commits. By analysing the code changes of the commits retrieved, the validity of a commit can be verified by finding the changes for vulnerable lines of code, as shown in **Figure 3.3**.

However, the step of correctly identifying the vulnerable lines of code is challenging as it varies with different programming languages. Firstly, different programming languages have different code syntax, and thus the code changes for addressing the same vulnerability might be different across different programming language. Secondly, the code changes of a commit only represent a small fragment of the whole source code, and there might be several code changes in different files that are addressing a same vulnerability.

Simple source code analysis technique might not be sufficient to find the relationship between code changes in different files.

The screenshot shows a code diff for the file `backend/src/main/java/com/benine/backend/database/MySQLDatabase.java`. The diff is titled "Use preparedStatement to prevent SQL Injection". It shows changes between version v1.0 and v0.9. The diff highlights the following changes:

```

232 52 - Statement statement = null;
    52 + PreparedStatement statement = null;
53 53   ResultSet resultSet = null;
54 54   try {
55 55   -   statement = connection.createStatement();
56 56   -   String sql = "SELECT tag_name FROM tagPreset WHERE preset_ID = "
57 57   -   + preset.getId();
58 58   -   resultSet = statement.executeQuery(sql);
    55 +   String sql = "SELECT tag_name FROM tagPreset WHERE preset_ID = ?";
    56 +   statement = connection.prepareStatement(sql);
    57 +   statement.setInt(1, preset.getId());
    58 +   resultSet = statement.executeQuery();

```

Figure 3.3: An example of code changes to prevent SQL injection [33].

3.5.3 Quality and Validity of Results

The analysis process is automated, which implies that the results returned might not fulfil the expectation. Automating the process of finding security vulnerabilities in open source repositories does not guarantee to produce a good result. In addition, manual evaluation is still required to check the performance of the tool. One constraint is that the automated process has to be exhaustively tested to find the optimal regular expression patterns. Although the tool might produce good results on some repositories, it does not indicate that the tool will produce consistent results on all repositories.

One major threat to the quality and validity of results is the amount of noise in real-world repositories. This project does not have a specially designed data set or a test data set, it uses real-world data set for calibration. The inconsistency of data in real-world data set will affect the quality of results. Therefore, the performance of the developed mining tool varies across different real-world repositories. There is no absolute certainty on the results produced, each commit found has to be inspected to verify its validity.

3.6 Testing

This section covers a brief overview of the testing stage. It will be necessary to consider some of the self-created test cases and scenarios in advance to find all possible bugs and flaws.

3.6.1 Unit Testing

Python provides a unit testing framework as part of its standard library, known as unittest [98], which offers a complete set of functions suffice to cover the unit testing of this project. Fundamental test cases include checking the functions for an expected result. Additional test cases are based on the functionality of the tool to cover every feature implemented.

Test Case	Expected Result	Status

Table 3.3: Documentation format of the test case.

3.6.2 System Testing

After completing the unit testing, the mining tool has to be tested for the functional requirements in **Table 3.1** on real-world repositories. Since the analysis process could run for several days continuously, the purpose of system testing is to ensure the stability of the whole analysis process, and to discover the edge cases that are not covered by unit testing yet. The test will also ensure that the program will be able to run on Python 3.6 and above under different operating systems and hardware specifications.

3.7 Evaluation

This section briefly discusses the approach to evaluate the mining tool on the real world projects to ensure that the requirements and criteria listed are practical and feasible.

3.7.1 Real-world Projects Evaluation

Real-world projects generally contain noise in their data due to inconsistency, incompleteness, and ambiguity [2], as shown in **Figure 2.1** and discussed in **Section 2.4.3**. Evaluating the mining tool on several real-world projects will test its ability of handling the noisy data. For the mining tool to be beneficial to the public, it must be able to produce results with a certain standard. This could be validated by verifying the accuracy and relevance of the results. It is presumed that the mining tool would only be suitable for a small set of repositories, and it might require comprehensive experiments of different configurations to achieve the best result.

Real-world projects including the Linux kernel [49], Apache HTTP Server [4], Apache Tomcat [7], Curl [22], OpenSSL [66], and Python programming language [80] are a good starting point for this project as they all have a large number of commits. This approach is reasonable as larger repositories are more likely to contain vulnerability- fixing commits and have a higher standard or informative commit messages.

3.7.2 Quality Evaluation

Having completed the testing stage does not infer that the repository mining tool would be practical in a real-world usage. To ensure the feasibility of this project, the tool has to be assessed by defining and measuring the quality metrics listed below:

- **Relevance:** The measurement of the number of relevant commits retrieved when given a regular expression that represent the commit message pattern of a vulnerability.
- **True positive rate:** The proportion of actual vulnerable commits or code being reported as a vulnerability.
- **False negative rate:** The proportion of vulnerable commits or code **not** being reported as a vulnerability.
- **Efficiency:** The total time taken required for the tool to complete the seaching process.

3.8 Ethical Issues

In this project, it is declared that any known or unknown vulnerabilities found by the mining tool in any repositories will not be publicly disclosed without the permission of the original authors. The reason is that publishing the vulnerabilities publicly would make the software highly vulnerable to attackers [8], and it is recommended to wait for the official announcement from the software vendors.

Chapter 4

Design

This chapter outlines the design concepts and justifies any decisions and approaches taken for the development of the project.

4.1 Programming Practices

To meet both functional and non-functional requirements defined in **Section 3.2**, a set of good practices has been adopted.

- **Performance:** Analysing large repositories can be time consuming. Therefore, the code should be written in a performance oriented style by following the advice of the Python Wiki [75]
- **Reliability:** Exception errors might occur during the runtime of the repository mining tool. Hence, exceptions should be handled in the code to ensure that the program continues to run when it encounters an error.
- **Simplification:** The coding logic should be clear and easy to follow. Large functions should be split into multiple sub-functions to improve code maintainability.
- **Documentation:** The code should be well documented and follow the PEP8 [74] coding style.

4.2 Implementation Model

There are several methods to implement the repository mining tool, and the selected approach must maximises the usability, while satisfying the requirements stated in **Section 3.2**. In this section, all approaches will be discussed and evaluated, with the selected approach further justified.

4.2.1 Considered implementation models

Model 1: Graphical User Interface (GUI)

The first model suggests implementing the repository mining tool with graphical user interface using libraries such as tkinter [97] or PyQt [79]. The interface can be divided into two parts:

1. **Repository Analysis:** This interface allows the users to open, view, and analyse the selected repository with specified parameters. During the analysis process, progress will be shown on the interface, and users are able to pause or stop the process.
2. **Statistical Evaluation:** This interface allows the users to read and edit the result file. Assuming the system has enough computing power and memory, the interface would allow reading and editing the result file concurrently with the analysis process. In this interface, users can browse through the results file and evaluate the results manually by marking the issues as false positive.

This model has several advantages, including better results presentation, multitasking ability on the same interface window, and easier result evaluation. However, the disadvantages includes the increased usage of system resources and incompatibility for batch job processing on the **High Performance Computing (HPC)** of the University of Sheffield.

Model 2: Executable Python Script

The second model suggests implementing the repository mining tool as an executable Python script. The tool does not have its own interface. It

must be executed in a **Command-line Interface (CLI)** such as a terminal, console, or shell. This model will divide the tool into two Python files:

1. **Repository Analysis:** This executable Python script file is responsible for analysing the repositories. It will accept user specified options via arguments, and perform the analysis on the command-line interface. Progress will be shown but the users are not able to pause the process. A **JSON** result file will be created when the analysis process has completed successfully.
2. **Statistical Evaluation:** This executable Python script file is responsible for analysing the result file. The script could be customised by the user to filter the results. The statistics is written to the **Standard Output (stdout)**, which defaults to the user's screen in the terminal.

In comparison with **Model 1**, this model features a lightweight and simplistic approach with all the core functionalities included. Running the tool as an executable script file uses less system resources and supports batch job processing on the **HPC** of the University of Sheffield. However, the users are not able to evaluate false positive results directly with the tool.

4.2.2 Chosen implementation model

After further consideration of both the software requirements and project objectives, **Model 2** was chosen to be the final implementation model. The reasons are justified below:

- Although new users generally find **GUI** to be visually intuitive, the tool itself does not require complex commands to operate. Moreover, the target users of the repository mining tool are researchers interested in the computer security field, which could be assumed to have basic knowledge of command line commands.
- Considering the time frame available for this project, creating a fully functional and visually attractive **GUI** might not be feasible. It would be more practical to allocate more time into improving the tool instead.
- The tool can analyse multiple repositories in a single command by

executing it in a shell script. This implies that the repositories analysis process could be divided into multiple parts and submitted to **HPC** for batch processing.

- Executing the Python script on a **CLI** uses less system resources than running a **GUI**.

4.3 Supported Programming Languages

Many open source repositories are written in more than one programming language. Therefore, the repository mining tool must be extensible and able to support multiple programming languages. The tool is predefined to support the repositories written in the following language:

- C/C++ (based on Flawfinder [101])
- Java (adapted and modified from graudit [52])
- Python (based on bandit [11])

In addition, the tool could be extended by the user to support other programming languages. The user must provide:

1. A rule set. This is a dictionary of common vulnerabilities. Each vulnerability are given a severity and confidence level.
2. A list of file extensions used by the programming language.
3. An optional regular expression pattern for non-essential lines. A non-essential line can be defined as a line of code that does not have positive contribution to the vulnerability analysis. Examples are code comments, blank line, and print statements for debugging purposes.

User-defined languages are expected to generate noisy results. This is because the tool does not understand the syntax and semantics of the code, thus it could not perform any control flow or data flow analysis. The generated results only give an indication of the possible vulnerabilities, it has to be reviewed and evaluated manually.

4.4 Overview of the Repository Mining Tool

4.4.1 Structure

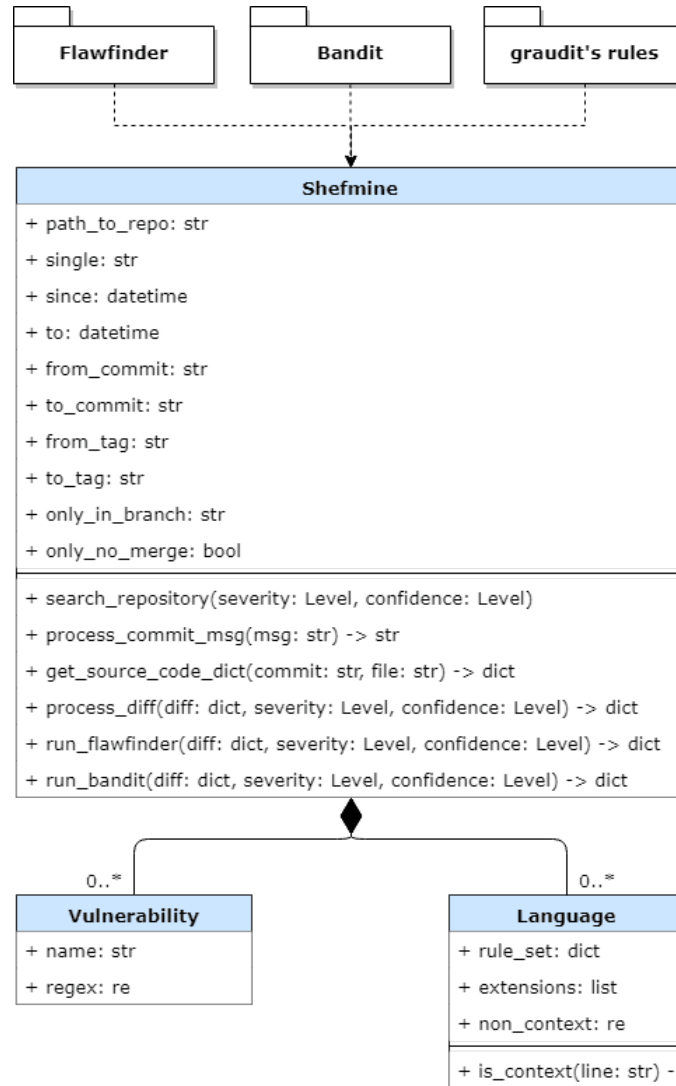


Figure 4.1: UML class diagram of Shefmine, the repository mining tool.

Figure 4.1 is a class diagram of the repository mining tool designed based on the requirements. The tool itself does not involve complex class relationships. Flawfinder and Bandit are external module dependencies, where the tool makes use of their analysis techniques to identify possible vulnerabilities.

4.4.2 Program Flow

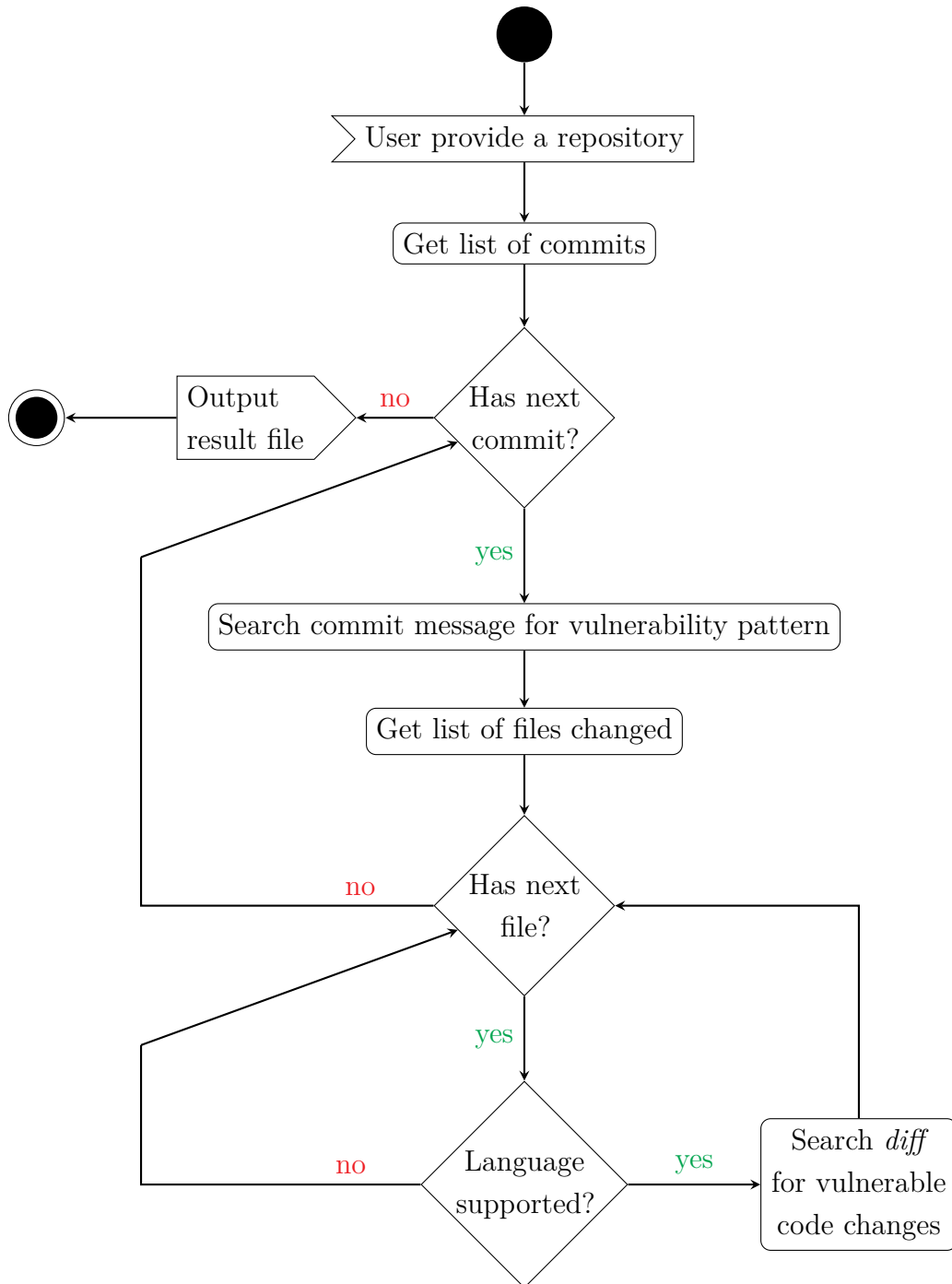


Figure 4.2: UML activity diagram of the repository mining tool

The program flow of the repository mining tool is shown in **Figure 4.2**. It is an general representation of the whole process. Hence, exhaustive details including exception handling, function calls, and user inputs validation are not included.

4.5 Commit Messages Matching

As mentioned in **Section 3.4**, there is already a concept of how commit messages matching should be functioning. It can be defined as the process of matching the commit message with the regular expression of each vulnerability pattern to find vulnerability-fixing commit. It requires some prior knowledge of the vulnerability patterns to define the regular expressions.

4.5.1 Vulnerability Patterns and Regular Expressions

The general idea of using regular expressions on commit messages is to construct the search queries with multiple conditions. Searching with regular expressions enables the tool to get results with one search, and avoid the usage of conditional statements to process the queries. Designing a specific, complete, and correct regular expression is challenging. This research question has two aspects to consider:

1. **Completeness:** If the objective was to achieve high completeness, then the regular expression would be designed to cover a broad range of string patterns. This would match more commit messages with the regular expression, which might possibly find more positive results. Similarly, false positive rate and the effort required for manual evaluation would increase.
2. **Correctness:** If the objective was to achieve high correctness, then the regular expression would be designed to be specific. This approach lowers the false positive rate, but increases the likelihood of generating false negatives.

The ideal design is to achieve high completeness and high correctness, but this assumption is not realistic. This is because the quality of commit

messages is not reliable, as discussed in **Section 3.5.1**. Achieving high correctness (low false positive rate) is desirable, but the additional effort in the improvement might not yield the corresponding improvements. Hence, the optimal approach would attempt to achieve high completeness first, then perform refinement on the regular expressions based on the results.

4.5.2 Expected Behaviour

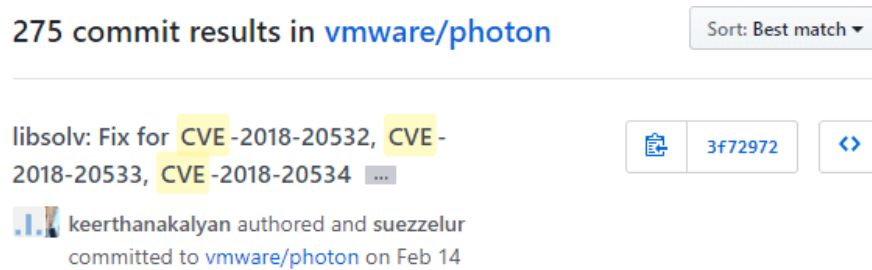


Figure 4.3: An example of a search query ‘CVE’ on GitHub.

Since regular expression is used instead of plain text search, the tool is aimed to perform at least, or better than the GitHub search in **Figure 4.3**. However, the tool does not produce a **GUI** output immediately after the search has completed, it stores the search results into a **JSON** file for manual evaluation.

4.6 Vulnerable Code Searching

Commit messages matching (**Section 4.5**) is not sufficient to prove the validity of the results. A commit message does not always summarise the actual changes in the commit object. To correctly identify a vulnerability-fixing commit, it is required to analyse the code changes.

4.6.1 Code Searching Techniques

There are several approaches to scan the source code for vulnerabilities, ranging from simple text matching to complex data flow analysis. While The

former is easier to build, and provides a quick way to find some potential security problems in the source code; the latter provides more accurate results, and it is able to detect complex vulnerabilities that involves nested control statements and function calls [17].

To select an approach, this research question has to consider several factors:

1. **Performance:** The amount of system resources and time required to complete one search of the chosen technique.
2. **Complexity:** The difficulty of implementing such technique.
3. **Feasibility:** The possibility measurement of producing a functional implementation of the chosen technique that satisfies the requirements.
4. **Accuracy:** The measurement of the ability to correctly identify the vulnerable lines of code.

4.6.2 Chosen Technique

After consideration of the factors listed in **Section 4.6.1**, it is chosen to implement a **static analysis technique**. It has been widely used in many security analysis tools including SpotBugs [96], Flawfinder [101], Bandit [11], and Progpilot [78]. As a result, it is required to discuss the strengths and weaknesses of the chosen technique to obtain an abstract indication of the expected results.

Strengths

- It scales accordingly to the code size. Unlike data flow analysis, it does not need to construct abstract syntax tree or control flow graph for its analysis process. Therefore, it is unlikely to encounter severe performance or memory issues.
- It is able to detect relatively simple vulnerabilities such as buffer overflows and SQL injections with high confidence. Although these vulnerabilities can sometimes be very complex as well.

Weaknesses

- It is unable to detect many security vulnerabilities automatically and accurately. These security vulnerabilities usually require sophisticated analysis method to detect.
- It is expected to generate numerous false positive results.
- It is difficult to prove the validity of the results. Since the analysis does not involve any data flow analysis, the reported results do not contain any detailed information such as variables value, and the steps to exploit the vulnerabilities.

4.6.3 Severity and Confidence Level

To mitigate the weaknesses, a severity and confidence level can be assigned to each vulnerability identified. They give the user an indication about the seriousness of a vulnerability, and the likelihood for the vulnerability to be correctly identified. With these parameters, the user can now choose to filter the results with severity or confidence higher than a certain level. This would lower the effort and time required for manual evaluation.

4.7 Level of Detail in Result File

Every Git commit object contains a lot of metadata [18]. Storing all metadata of each commit found will consume a substantial amount of storage space, as well as increasing the reading time of the result file. Hence, only relevant information that will be used in statistical evaluation are included, which are listed below:

1. The commit message (optional)
2. The date of the commit
3. The matching groups of regular expression (**Section 4.5**)
4. The vulnerability pattern of the matched regular expression (**Section 4.6**)

5. The vulnerable code changes identified, with line number, code severity level, and confidence level. (**Section 4.6**)

The commit message is just for the ease of user reference, which might be trivial as it does not have positive usage for the analysis but storing it will increase the file size.

4.8 Statistical Evaluation

The final part of this project is to create a statistical analysis tool to analyse the result file produced. The purpose to calculate these statistics is to study the relationship among vulnerability-fixing commit, commit message, and lines changed. The proposed method is to load the **JSON** result file into a Python dictionary first, then calculate the statistics. Below is a list of statistics to calculate in the statistical analysis tool:

1. Total commits found
2. A list of years that represents the number of commits found in each of the year
3. A list of vulnerability types found by regular expressions match and vulnerable code search
4. The number of commits found, calculated according to severity and confidence level
5. The number of commits found that are **only** matched by regular expressions
6. The number of commits found that **only** have vulnerable lines changed
7. The number of commits found that have **both** regular expressions match and vulnerable lines changed

Chapter 5

Implementation and Testing

This chapter will discuss the implementation of the repository mining tool based on the specifications described in previous chapters. It will also provide insights into the challenges encountered during the implementation.

5.1 Project Environment and Structure

5.1.1 Hardware Requirements

This project relies on strong processing power and high availability of memory (RAM). The speed of the analysis process is determined by the processing power of the CPU. For large repositories, it is recommended to allocate at least 4GB of RAM to the process, otherwise the process might encounter a memory error and terminate.

5.1.2 Software Requirements

As concluded in **Section 3.3.1**, the project has been implemented in Python 3. There are several Python 3 versions available, and the project used the latest stable version at the time of this project, which is Python 3.7.3. Specifically, the source code is written with the new syntax and features introduced in the latest version, which implies that the older versions of Python 3 are not able to run it before updating.

5.1.3 Structure

Figure 5.1 shows the directory structure of the project. It follows the design in **Figure 4.1**, where the file `vulnerability.py` and `language.py` contain the class `Vulnerability` and `Language` respectively. It is not necessary to create an individual file for each programming language in the `languages` folder, but it is recommended in practice for better maintainability and clarity of the project.

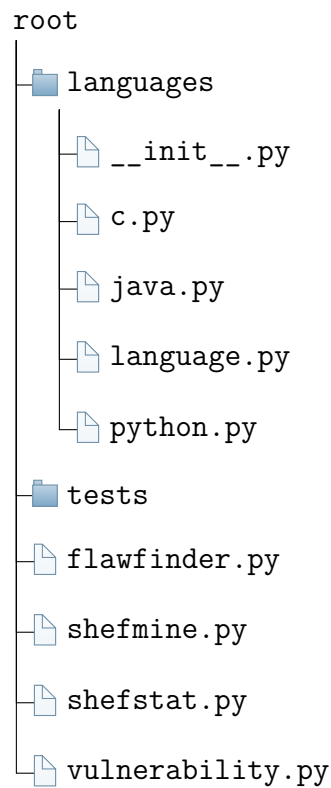


Figure 5.1: Directory structure of the project

5.2 Commit Messages Matching

This section is an explanation of how the design defined in **Section 4.5** was implemented. The implementation is split into three stages, and each subsection below represent one of the stage.

5.2.1 Vulnerabilities

The vulnerabilities represent the type of vulnerability-fixing or vulnerability-inducing commit to find. Each vulnerability is an instance of the class `Vulnerability`, and has a regular expression pattern that indicates the possible string format of the commit messages. The vulnerabilities below are implemented into the tool:

- Broken Access Control
- Broken Authentication and Session Management
- Buffer Overflow
- Bug Tracker Issue
- Context Leaks
- Cross-Site Request Forgery
- Cross-Site Scripting
- Distributed Denial-of-Service / Denial-of-Service
- Encryption Issues
- Hard Coded
- Injection
- Insufficient Attack Protection
- Memory Leaks
- Miscellaneous
- Null Pointers
- Overflow
- Resource Leaks
- Path / Directory Traversal
- SHA-1 Collision
- Security Misconfiguration
- Sensitive Data Exposure
- Using Components with Known Vulnerabilities
- Underprotected APIs

A new vulnerability can be created by passing two arguments to the `Vulnerability` class: the name of the new vulnerability, and the regular expression pattern that describes the commit message pattern, as demonstrated in **Listing 5.1**.

```
Vulnerability(
    'Distributed Denial-of-Service / Denial-of-Service',
    '(dos|((distributed)? denial.*of.*service)|ddos|deadlocks?)'
```

Listing 5.1: Creating a new instance of `Vulnerability`.

5.2.2 Writing the Regular Expressions

Having defined the list of vulnerabilities to support, the next step is to write the regular expression for each vulnerability. This is the most challenging part in commit message matching as there are no specific definitions for the commit message format. Achieving **completeness** and **correctness** is infeasible, it is impracticable to construct a regular expression that will cover all the possible cases of a commit message.

The best mitigating approach is to assume that the commit messages of the same vulnerability type would be having a similar format, or sharing some parts of a string. This could be accomplished with the capturing group and wildcard character of regular expression. The graphical representation of the regular expression `(fix|rem|patch|found|prevent).* mem.* leak|mem.* leak (fix|removed?|patch|found|prevent)` is shown in **Figure 5.2**.

In real-world repositories, the regular expressions have a high possibility of matching a false positive result. As discussed in **Section 3.2**, the quality of commit message is not reliable. This factor has been considered into the final evaluation of the results.

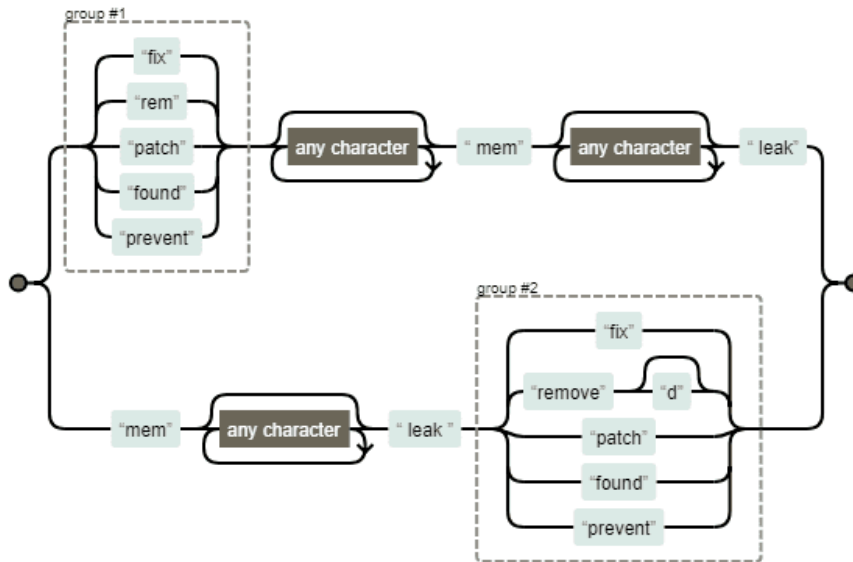


Figure 5.2: An example of using capturing groups and alternation in regular expression.

5.2.3 Searching for the Matches

The implementation for the search algorithm is not difficult since all regular expression operations are already provided in the Python's `re` [81] library. It is only required to ensure that the operations will return the expected results for a given regular expression.

The algorithm is a single conditional statement. Given a commit message and a regular expression pattern, if the commit message matches the regular expression pattern, then include the commit into the result. As shown in **Listing 5.2**, the vulnerability name and matched string are saved into the result file, which satisfied the requirements in **Section 4.7**.

```
"de072100775cc29e6cd93a75466cecbd1086f258": {  
    ...  
    "vulnerabilities": [  
        {  
            "name": "Buffer Overflow",  
            "match": "Fixed buffer overflow"  
        }  
    ],  
    ...  
},  
"9790a2706573359e02fcfc5f18f9907467f4ec49": {  
    ...  
    "vulnerabilities": [  
        {  
            "name": "Sensitive Data Exposure",  
            "match": "Fix for #1303614 and #1174712:\n- __dict__ des"  
        }  
    ],  
    ...  
}
```

Listing 5.2: A partial snippet of the result file of CPython [80] repository.

The first commit in **Listing 5.2** is a successful example of the search, but the second commit is less specific and might require a manual evaluation. In the second commit, the string ‘for #1303614 and #1174712:\n__dict__’ is a wildcard match. Although it is identified that the second commit is a fix for some issues, the actual vulnerability fix might not be a **Sensitive Data Exposure** type.

5.3 Vulnerable Code Searching

After commit message matching, the next step is to analyse the code changes in the commits. This section will discuss the technique used to extract the information from the Git commit object, and the use of the extracted information to find potential vulnerabilities in Git commit.

5.3.1 Types of the Code Changes

All code changes are recorded in the commit, and these changes can be categorised into added, deleted, and unchanged.

- **Added:** If an added line is reported as vulnerable, then the commit is identified as a **vulnerability-inducing** commit.
- **Deleted:** If a deleted line is reported as vulnerable, then the commit is identified as **vulnerability-fixing** commit.
- **Unchanged:** If an unchanged line in a changed file is detected as vulnerable, then it is possible for the repository to contain vulnerabilities that are not yet known.

It is important to note that the objective of the repository mining tool is to find potential vulnerabilities through the commit history. Therefore, if a vulnerability is introduced at the very first commit, and the file that contains the vulnerability has never been modified after that, then it would not be possible for the tool to find it.

5.3.2 Flawfinder and Bandit Integration

The Flawfinder program is a standalone tool and consists of only one file, but Bandit has a more complex structure and consists of many classes. They were both written in Python and support at least Python 2.7 and Python 3. Both of them are imported into the main file as modules, and their functions can be directly invoked. No compatibility issues were countered during the integration process.

Both Flawfinder and Bandit take a list of source files or directories as input from the command-line, thus it is not able to pass the source code in a string form directly into the functions. In GitPython [35], the Diff object of a commit stores the old and new source code as binary streams. The integration steps are:

1. Decode the binary streams into string.
2. Write the decoded source code into a temporary file.
3. Pass the temporary file into the relative Flawfinder or Bandit function.
4. The results will be returned when the Flawfinder or Bandit has finished the analysis.

5.3.3 User-defined Programming Language

Python is an **Object-oriented Programming (OOP)** language, it can fulfil the design specifications of **Section 4.3**. To support an extra programming language, an instance of the **Language** class must be created. The ‘constructor’ (the `__init__` method) takes three parameters, a rule set, a list of file extensions, and an optional regular expression object for non-essential lines.

The importance and impact of the three parameters on the analysis results are explained below:

1. **Rule set:** This is related to the core research objectives of this project. The repository mining tool will scan the source code with each rule in the rule set and record the matching one. A well defined rule set could

reduce the possibility of false positives. However, there is a still limit on the effectiveness of it as the tool is only performing pure regular expression search with these rules on the source code. An example of the structure of a rule set is shown in **Listing 5.3**.

```
ruleset = {  
    'createStatement\s*\(.*\)': {  
        'severity': 'HIGH',  
        'confidence': 'LOW'  
    },  
    ...  
}
```

Listing 5.3: An example snippet of the Java ruleset.

2. **File extensions:** Some programming languages use multiple file extensions as their source code file. The purpose of this parameter is to prevent the tool from carrying the search on irrelevant files.
3. **Regular expression object for non-essential lines:** The purpose of this parameter is to improve the accuracy and performance of the tool. By excluding the non-essential lines from the analysis process, the tool is able to perform faster, and it is less likely to obtain false positive results. If this parameter is not provided, then the tool will analyse every single line of the source code.

A rule can be either a plain string or a regular expression pattern, the algorithm will take the rule and perform a word boundary search (`\b{rule}\b`). In contrast to Flawfinder and Bandit, this algorithm does not need to parse the source code and tokenise them in advance. With these justifications, it is predicted that the tool would not have better performance than Flawfinder or Bandit (**Section 5.3.2**).

5.4 Configurable Options

As specified in **Section 4.2.2**, the repository mining tool is implemented as an executable Python script. This indicates that the analysis process is

customisable with command-line arguments.

- The branch to analyse.
- The commit range to analyse. This can be specified by providing two dates, two tags, or two commit hash.
- The minimum severity level.
- The minimum confidence level.

5.5 Exceptions and Errors

An unhandled exception will cause the program to terminate. Exception occurs whenever syntactically correct Python code results in an unhandled error. Below is a list of errors encountered during the implementation and testing. The list does not include the errors that are already handled by Flawfinder and Bandit.

- **Decode error:** Not all binary streams are encoded in the same encoding. This error will occur when it is unable to decode the binary streams due to the encodings used is not supported by standard Python codecs.
- **Memory and storage error:** This error will occur when the tool is trying to output a very large result file but the memory and storage space allocated is not enough.

5.5.1 Special Case

There are cases where the analysis progress is completely frozen. For example, the commit [b9a311](#) has 17925 files changed, which will cause the progress to be stucked for a long time. Therefore, the tool is programmed to skip a commit if it has more than 100 files changed.

5.6 Testing

The system was tested both automatically and manually to ensure that each feature has functioned correctly. Automatic testing covered the test cases written using the Python’s unit testing library [98]. Manual testing mostly covered some edge cases and tried to reproduce some rare errors to ensure that they will be handled by the system.

5.6.1 System Environment

Test Case	Expected Result	Status
Windows 10, Python 3.6 and above	The program runs	Pass
Windows 10, Python 3.5 and below	Syntax error, program does not run	Pass
Linux, Python 3.6 and above	The program runs	Pass
Linux, Python 3.5 and below	Syntax error, program does not run	Pass
macOS, Python 3.6 and above	Syntax error, program does not run	Pass
macOS, Python 3.5 and below	Syntax error, program does not run	Pass

Table 5.1: Test cases of system environment.

5.6.2 Program Initialisation

Test Case	Expected Result	Status
Providing an invalid path	Exception is handled and error message is printed	Pass
Providing an invalid Git repository	Exception is handled and error message is printed	Pass
Specifying an invalid branch	Exception is handled and error message is printed	Pass

Specifying an invalid commit hash	Exception is handled and error message is printed	Pass
Specifying an invalid commit range	Exception is handled and error message is printed	Pass
Providing valid options	Program starts to analyse	Pass

Table 5.2: Test cases of program initialisation.

5.6.3 Repository and Commit Properties

Test Case	Expected Result	Status
Providing an empty Git repository	Analysis started but no result file output	Pass
Analysing an empty commit	The commit is skipped	Pass
Analysing a commit with binary files changed only	Binary files are excluded from analysis	Pass
Analysing a commit with more than 100 files changed	The commit is skipped	Pass
Decoding a file changed that has invalid characters	The characters are ignored, and analysis continues to run	Pass
Decoding a file changed of non-supported encoding	The file is skipped	Pass

Table 5.3: Test cases of repository and commit properties.

5.6.4 Commit Message Matching

Test Case	Expected Result	Status
Given a non-matching string and a regular expression	The search operation returned None	Pass
Given a matching string and a regular expression	The search operation returned the correct matched groups	Pass

Table 5.4: Test cases of commit message matching.

5.6.5 Vulnerable Code Searching

Test Case	Expected Result	Status
File extension is not supported	The file is skipped	Pass
File is empty	No issues returned	Pass
Analysing a commit that has vulnerable code added	Vulnerabilities is saved into the 'added' list	Pass
Analysing a commit that has vulnerable code deleted	Vulnerabilities is saved into the 'deleted' list	Pass
Analysing a commit that has vulnerable code unchanged	Vulnerabilities is saved into the 'unchanged' list	Pass

Table 5.5: Test cases of vulnerable code searching.

5.6.6 Performance Testing

- **Control variables:** The hardware specifications
- **Independent variables:** The number of commits to analyse, the size of a commit (number of files changed, number of code changes)
- **Dependent variables:** The time taken to complete the analysis

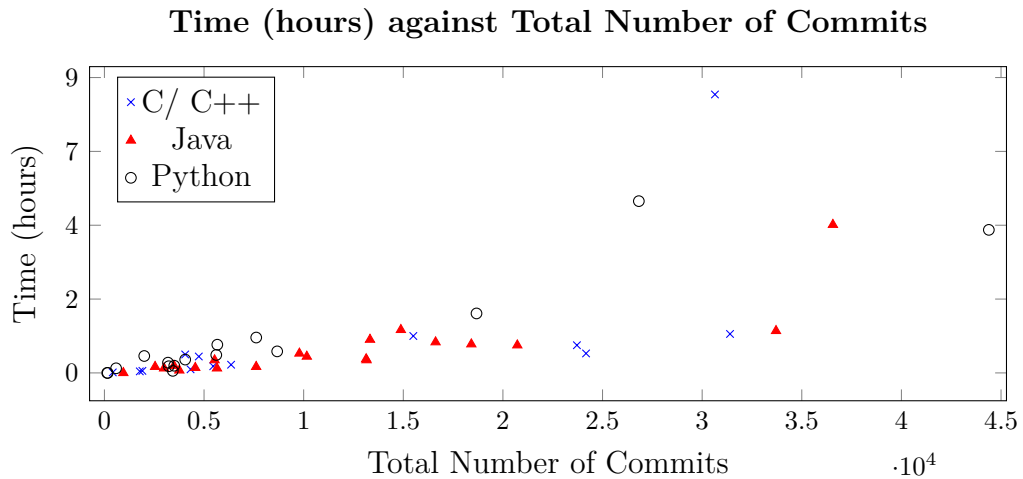


Figure 5.3: Number of potentially vulnerable commits found since 1990 in all repositories analysed.

According to **Figure 5.3**, it has been shown that there is a positive correlation between the time taken and the total number of commits. The details of the results are recorded in **Table A1**.

Overall, the performance of the repository mining tool is considered fast after taking into account that all files changed of the whole commit history are analysed in the testing. In a practical scenario, the users might only be interested in the commits that newer than a specific date or tag. If these options are specified, then the time taken would decrease as there are less commits to analyse.

Chapter 6

Results and Discussion

This chapter will present the main results of the tool, together with the evaluation. The critical discussion will present a detailed analysis of the results, as well as the possible improvements to be made.

6.1 Data Set

The analysis data set consists of 52 open source repositories (**Table A1**) selected from GitHub [34]. Specifically, 15 of these repositories are C/C++ based, 20 of them are Java based, and 17 of them are Python based. These repositories are selected based on the actual usage of large software vendors and their user base.

6.2 C/C++ Repositories

Table 6.1 has shown the 5 most common vulnerability types found by **commit message matching**. These most common vulnerability types, however, do not represent the actual security condition of the overall C/C++ repositories. In particular, **Null Pointers** (7397 found, 6.34%), **Memory Leaks** (5242 found, 4.49%), **Overflow** (4486 found, 3.85%), and **Buffer Overflow** (1286 found, 1.10%) are more relevant issues in the C/C++ programming language.

Vulnerabilities	Total	Percentage
Bug Tracker Issue	29,884	25.62%
Sensitive Data Exposure	17,448	14.96%
Encryption Issue	17,022	14.59%
Miscellaneous	12,741	10.92%
Null Pointers	7,397	6.34%

Table 6.1: Top 5 vulnerabilities matched by the regular expressions in the C/C++ repositories analysed.

6.2.1 Case Study

Linux

The Linux repository [49] has the most commit counts in the data set. The commit messages are well-documented, and most of the bug-fixing commits contain a reference link to a specific bug tracker issue. The partial statistics shown in **Table 6.2** are more specifically targeted to the repository, and are considered to be a good research objective for future references.

Vulnerabilities	Total	Percentage
Bug Tracker Issue	18,886	21.89%
Null Pointers	7,073	8.20%
Security Misconfiguration	5,867	6.80%
Distributed Denial-of-Service / Denial-of-Service	5,725	6.64%
Encryption Issues	5,242	6.08%
Memory Leaks	4,223	4.89%
Overflow	4,052	4.70%
Underprotected APIs	3,967	4.60%
Injection	1,391	1.61%
Hard Coded	1,308	1.52%
Buffer Overflow	1,114	1.29%
	<u>58848</u> 86283	68.20%

Table 6.2: Partial statistics of the Linux repository analysis.

6.3 Java Repositories

Vulnerabilities	Total	Percentage
Bug Tracker Issue	4,304	24.25%
Security Misconfiguration	3,468	19.54%
Encryption Issue	3,339	18.82%
Sensitive Data Exposure	2,674	15.07%
Distributed Denial-of-Service / Denial-of-Service	849	4.78%

Table 6.3: Top 5 vulnerabilities matched by the regular expressions in the Java repositories analysed.

The statistics in **Table 6.3** do not reflect the actual condition as well. Majority of the Java repositories analysed are server or networking based projects. Hence, vulnerabilities such as **DDoS / DoS** (849, 4.78%), **Underprotected APIs** (695, 3.92%), **Injection** (203 found, 1.14%), and **Cross-Site Scripting** (198 found, 1.12%) are more specific and relevant to the data set.

6.3.1 Case Study

Apache Tomcat

Vulnerabilities	Total	Percentage
Bug Tracker Issue	2,410	62.00%
Encryption Issues	699	17.98%
Sensitive Data Exposure	327	8.41%
Distributed Denial-of-Service / Denial-of-Service	101	2.60%
Miscellaneous	82	2.11%
Memory Leaks	58	1.49%
Security Misconfiguration	37	0.95%
Underprotected APIs	35	0.90%
Cross-Site Request Forgery	19	0.49%
	$\frac{3768}{3887}$	96.94%

Table 6.4: Partial statistics of the Tomcat repository analysis.

In contrast to the Linux repository, the commit messages of Tomcat repository [7] are less informative but still provide enough context as a whole. It is a server implementation project, thus it is expected to find more network related vulnerabilities in this repository. However, this assumption implies that the statistics in **Table 6.4** might not be accurate. The problem is further inspected in **Section 6.6**.

6.4 Python Repositories

Vulnerabilities	Total	Percentage
Encryption Issues	3,930	35.02%
Sensitive Data Exposure	2,535	22.29%
Miscellaneous	1,028	9.16%
Underprotected APIs	704	6.27%
Security Misconfiguration	675	6.02%

Table 6.5: Top 5 vulnerabilities matched by the regular expressions in the Python repositories analysed.

Similarly, the statistics in **Table 6.5** do not represent the overall condition of all Python repositories. The validity of the statistics has to be evaluated before concluding it.

6.4.1 Case Study

CPython

CPython [80] is the repository of the Python programming language. The commit messages strictly follow a specific documenting guideline, and are considered to have the same quality as the Linux repository. According to the repository page on GitHub, 61.4% and 32.5% of source code is written in Python and C respectively. This explains why **Table 6.6** shows a mixed range of vulnerability types. Moreover, the repository also contains numerous files that are implemented as the built-in modules for end users.

Vulnerabilities	Total	Percentage
Encryption Issues	1,251	30.04%
Sensitive Data Exposure	911	21.88%
Miscellaneous	332	7.97%
Overflow	330	7.93%
Memory Leaks	288	6.92%
Underprotected APIs	245	5.88%
Distributed Denial-of-Service / Denial-of-Service	234	5.62%
Security Misconfiguration	208	5.00%
Bug Tracker Issue	133	3.19%
Null Pointers	77	1.85%
	$\frac{4009}{4164}$	96.28%

Table 6.6: Partial statistics of the Tomcat repository analysis.

6.5 Results Summary

This section presents a summary of the results. The results presented is the combined total of statistics from the analysis results of C/C++, Java, and Python repositories. The main purpose is to show an overall trend of the results, and provide a brief perspective on the overall results.

Vulnerabilities	Total	Percentage
Bug Tracker Issue	34,799	23.90%
Encryption Issue	24,291	16.68%
Sensitive Data Exposure	22,657	15.56%
Miscellaneous	14,409	9.89%
Security Misconfiguration	10,457	7.18%

Table 6.7: Top 5 vulnerabilities matched by the regular expressions in all repositories analysed.

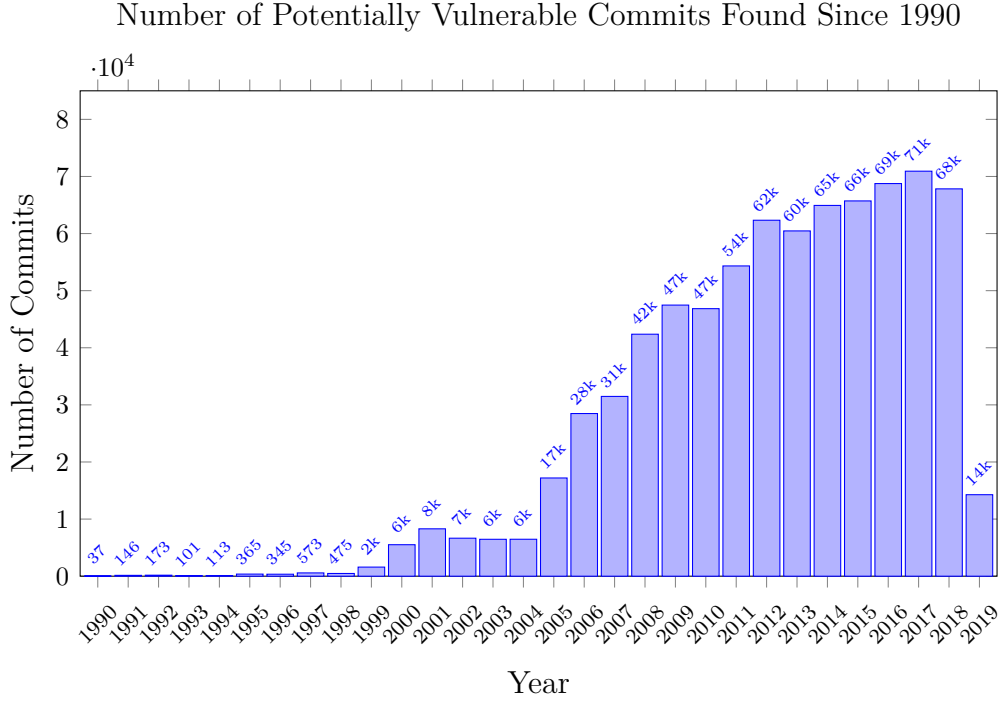


Figure 6.1: Number of potentially vulnerable commits found since 1990 in all repositories analysed.

Some repositories have longer commit history, while some of them have more commit counts than others. This applies to the number of vulnerable commits found in each repository as well. According to **Figure 6.1**, the number of potentially vulnerable commits found has increased over the years. A potentially vulnerable commit can be either a vulnerability-inducing or vulnerability-fixing commit, as explained in **Section 5.3.1**. This can be explained by the increase of security awareness among developers, and the increased number of Internet users over the past 20 years. Therefore, it could be assumed that both the number of security issues and the number of vulnerability-fixing commits will be increasing in the future.

It can be observed in **Figure 6.1** that the number of potentially vulnerable commits has started to grow rapidly since 2004, which is the same year as the initial release of **OWASP** Top Ten project [71]. Hence, the sudden grow from 2004 to 2005 could be assumed as the increase of security fixes

being published by the developers. If this assumption was true, then the correctness of the repository mining tool could be proved to a certain extent.

6.6 Evaluation

This section will assess the validity of the results, and show the problems discovered during the evaluation process. Lastly, the achievements of this project will be evaluated, and future improvements will be discussed.

6.6.1 Threats to Validity of Results

There are several threats identified during the results collection and evaluation process. These threats may potentially affect the final accuracy of the results, and lower the overall quality of the repository mining tool.

- **Ambiguous commit messages:** The statements in **Section 3.5.1** have been proved to be correct. Many commits do not provide a clear and precise summary of the changes introduced. This might cause the actual vulnerability-fixing commits to be classified as false negatives.
- **Problematic commit properties:** Git allows empty commit to be made, as well as setting the commit date to any date. Commits with empty message or no code changes will require extra effort for verification. For example, [224426f](#), [09f2724](#), and [12ca45f](#) have incorrect date with respect to their parent commit, such issue will affect the results in **Figure 6.1**.
- **Hidden information:**
 - The commit [df7edb](#) is a fix for CVE-2019-0211 [26], but the commit message did not mention the CVE identifier.
 - The commit [9233d9](#) ([GitHub link](#), [Subversion link](#)) of Tomcat repository is an actual fix for CVE-2017-5651. However, the commit message of this commit does not contain the CVE identifier in the GitHub version, whereas the Subversion version does. It is unknown whether the removal of CVE identifier from the commit

message in the GitHub version was intentionally, or a conversion error from Subversion to Git.

- **Existence of noise in code changes:** This factor have already been considered in **Section 3.5.3**, but the amount of noise was beyond the expectation. It was later noticed that the precautions method implemented were not able to exclude various non-essential code changes.
- **Human error:** Wrong judgements might be made during the manual validation due to insufficient knowledge in security vulnerabilities and limited proficiency in programming languages.

6.6.2 Problems and Limitations

Commit Message Matching

According to **Table 6.1**, **Table 6.3**, **Table 6.5**, and **Table 6.7**, the following vulnerability types are commonly being matched: **Bug Tracker Issue**, **Encryption Issue**, **Sensitive Data Exposure**, and **Miscellaneous**. This scenario is theoretically possible, but not very likely when considering the actual data set used. The high proportion of these vulnerability types in the overall results does not appear to be logical, especially when some of them still remain with high percentage in the case study.

Upon manual inspection, it is discovered that the majority of commits matched by the regular expressions are positive results, but many of them are misclassified. Therefore, it can be concluded that some of the regular expressions were broadly defined, while others were strictly defined. For example, the commit message of [89fd8d](#) is matched as Sensitive Data Exposure, but the code changes are all comment improvements. The impact could be either significant or minor, which depends on the proportion of positive results in the misclassified commits.

Vulnerable Code Searching

As mentioned in **Section 6.6.1**, the repository mining tool could not detect and exclude all non-essential code changes from the analysis. This limitation

is less significant in Flawfinder and Bandit implementation, but more severe in the implementation for user-defined programming language. Moreover, the tool did not ignore whitespace changes (e.g. `memcpy()` to `memcpy ()`), which then introduced considerable amount of false positives into the results.

One major limitation of the repository mining tool is the inability to detect very specific vulnerability fixes. This is also a known limitation in many static analysis tools. For instance, the commit [cd2b7a](#) of Apache HTTPD repository contains CVE identifier in the commit message, but the tool was unable to detect the actual code changes that addressed the vulnerability fix. In other cases, vulnerability-fixing code changes might also be distributed to several files, which would increase the difficulty for such tools to detect it.

6.6.3 True Positive Rate

To calculate the true positive rate, the commits found must be evaluated manually. The sampling process must be completely stochastic to minimise human bias. For each programming language, 3 random repositories were randomly chosen. Then for each repository chosen, a maximum of 4 commits from each category below was randomly chosen for manual evaluation. The minimum severity and confidence level for vulnerable code changes are set to high for the best result.

- Only regular expression match
- Only vulnerable code changes
- Only added vulnerable code changes
- Only deleted vulnerable code changes
- Only added and deleted vulnerable code changes
- Both regular expression match and vulnerable code changes

120 out of 271 commits were found to be vulnerability-fixing commits. The sample size might be relatively small compared to the total commits found, but the commits evaluated were randomly selected. Such commits give a general indication of the overall results. Hence, the true positive rate of the

repository mining tool is approximately **44.28%**. Additionally, the following situations were observed during the evaluation process:

- If a commit contains both regular expression match and vulnerable code changes, then it is likely to be a vulnerability-fixing commit.
- The test files of some repositories includes running the vulnerable code, which is very likely to be detected and introduced false positives.
- Some regular expressions are able to perform unexpectedly well on matching ambiguous commit messages.

6.7 Goals Achieved

6.7.1 Objectives

The primary objective of this project is to find security issues in open source software repositories through a series of analysis processes. With a true positive rate of 44.28%, the project has proved its practicability, and demonstrated a feasible way to achieve the objective. Although the current implementation faces many problems and limitations, the repository mining tool was able to reach the goal, and complete the analysis process successfully.

6.7.2 Functional Requirements

Criteria	Importance	Status
Compatibility	Essential	Achieved
Completeness	Essential	Partly Achieved
Informative	Essential	Achieved
Repeatable	Essential	Achieved
Scalability	Essential	Achieved
Automated Evaluation	Desirable	Achieved
Extensibility	Desirable	Achieved
Robustness	Desirable	Achieved

Table 6.8: Achievement status of functional requirements in **Table 3.1**.

The completeness criteria is partly achieved due to the high false positive rate and the inconsistency in finding hidden vulnerability-fixing commits.

6.7.3 Non-functional Requirements

Criteria	Importance	Status
Code Style	Essential	Achieved
Documentation	Essential	Achieved
Lightweight	Essential	Achieved
Open source	Essential	Achieved
Performance	Desirable	Achieved

Table 6.9: Achievement status of non-functional requirements in **Table 3.2**.

6.8 Further Work

1. **Regular expressions improvement:** This is an iterative process. The refinement has to be carried out in stages to ensure that the patterns are specific and do not overlap with each other's domain.
2. **Noise elimination improvement:** Implements a more precise technique by following the approach of Kawrykow and Robillard [44].
3. **Performance optimisation:** The current implementation is limited to using a single processor core. The program could be extended to make use of the Python built-in multiprocessing [59] module to run the program on multiple processors for simultaneous execution. This improvement is expected to lower the analysis time significantly.
4. **Ability to locate the origin of the vulnerabilities:** The tool should be improved to link the vulnerability-fixing commit with a correct vulnerability-inducing commit.
5. **Automatic validation of bug tracker issues:** For every bug tracker issue referenced in the commit message, the tool could verify it by sending a HTTP request to the link and analyse the HTML file retrieved.

Chapter 7

Conclusion

This project began with research into the security of open source repositories, which led into a deeper analysis of the techniques used in **Mining Software Repositories (MSR)**. During the research period, a wide range of options were explored, and the most suitable choices were selected and adapted into the implementation. The research has helped to recognise several limitations and difficulties, which also helped to reduce the number of vain attempts being made during the implementation.

After the research has been completed, the project objectives were set and the requirements were outlined. Following the requirements, the selection of software was reviewed, and the reasoning of the final decisions was justified. The project was then assessed for its feasibility by considering the limitations of the chosen approach and the factors that might affect the quality of analytical results. The testing and evaluation were planned in details to cover all aspects of functionality.

The repository mining tool was designed and implemented with consideration. The chosen approaches were further explained with their strenghts and weaknesses presented and compared to alternative methods. The implementation followed the design specifications and introduced several improvements during the process. The final version of the tool developed was tested with unit testing and system testing and passed all the test cases.

Overall, the repository mining tool was found to have a true positive rate of 44.28%. Several problems and limitations were identified in the manual evaluation, and they can be corrected and improved. In conclusion, the result was satisfying but there is still room for improvement. The tool produced has achieved the implementation objectives and satisfied the software requirements in **Section 3.2**. This project has demonstrated its practicability and the possibility of achieving such research objectives in related fields.

Acronyms

APIs application processing interfaces. 1

CLI Command-line Interface. 29, 30

CVE Common Vulnerabilities and Exposures. viii, 2, 6–8, 12, 19, 34

CWE Common Weakness Enumeration. 6

FLOSS Free/Libre and Open Source Software. 1

GUI Graphical User Interface. 28–30, 34

HPC High Performance Computing. 28–30

JSON JavaScript Object Notation. 3, 17, 20, 21, 29, 34, 37

MSR Mining Software Repositories. 9, 13, 14, 62

NVD National Vulnerability Database. 1

OOP Object-oriented Programming. 44

OWASP Open Web Application Security Project. 2, 7, 8, 56

stdout Standard Output. 29

XML Extensible Markup Language. 20

Bibliography

- [1] A. Alali, H. Kagdi and J. I. Maletic, ‘What’s a typical commit? a characterization of open source software repositories’, in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, IEEE, Jun. 2008, pp. 182–191. DOI: 10.1109/ICPC.2008.24.
- [2] S. S. Alqahtani, E. E. Eghan and J. Rilling, ‘Tracing known security vulnerabilities in software repositories—a semantic web enabled modeling approach’, *Science of Computer Programming*, vol. 121, pp. 153–175, 2016, ISSN: 0167-6423. DOI: 10.1016/j.scico.2016.01.005.
- [3] N. Antunes and M. Vieira, ‘Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services’, in *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, IEEE, Nov. 2009, pp. 301–306, ISBN: 978-0-7695-3849-5. DOI: 10.1109/PRDC.2009.54.
- [4] *Apache http server repository*. [Online]. Available: <https://github.com/apache/httpd> (visited on 31/10/2018).
- [5] *Apache httpd git commit*. [Online]. Available: <https://github.com/apache/httpd/commit/64d44> (visited on 10/04/2019).
- [6] *Apache tomcat git commit*. [Online]. Available: <https://github.com/apache/tomcat/commit/6d3c11> (visited on 10/04/2019).
- [7] *Apache tomcat repository*. [Online]. Available: <https://github.com/apache/tomcat> (visited on 31/10/2018).

BIBLIOGRAPHY

- [8] A. Arora, R. Krishnan, R. Telang and Y. Yang, ‘An empirical analysis of software vendors’ patch release behavior: Impact of vulnerability disclosure’, *Information Systems Research*, vol. 21, no. 1, pp. 115–132, 1st Mar. 2010. DOI: 10.1287/isre.1080.0226.
- [9] A. Arora and R. Telang, ‘Economics of software vulnerability disclosure’, *IEEE security & privacy*, vol. 3, no. 1, pp. 20–25, 14th Feb. 2005, ISSN: 1540-7993. DOI: 10.1109/MSP.2005.12.
- [10] D. Balzarotti, M. Monga and S. Sicari, ‘Assessing the risk of using vulnerable components’, in *Quality of Protection*, D. Gollmann, F. Massacci and A. Yautsiukhin, Eds., Springer, 2006, pp. 65–77, ISBN: 978-0-387-36584-8. DOI: 10.1007/978-0-387-36584-8_6.
- [11] *Bandit is a tool designed to find common security issues in python code*. Python Code Quality Authority. [Online]. Available: <https://github.com/PyCQA/bandit> (visited on 01/03/2019).
- [12] V. R. Basili and B. T. Perricone, ‘Software errors and complexity: An empirical investigation’, *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1st Jan. 1984, ISSN: 0001-0782. DOI: 10.1145/69605.2085.
- [13] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak and D. Engler, ‘A few billion lines of code later: Using static analysis to find bugs in the real world’, *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010. DOI: 10.1145/1646353.1646374.
- [14] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov and P. Devanbu, ‘Fair and balanced?: Bias in bug-fix datasets’, in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE ’09, Amsterdam, The Netherlands: ACM, 24th Aug. 2009, pp. 121–130, ISBN: 978-1-60558-001-2. DOI: 10.1145/1595696.1595716.
- [15] L. C. Briand, V. R. Brasili and C. J. Hetmanski, ‘Developing interpretable models with optimized set reduction for identifying high-risk software components’, *IEEE Transactions on Software Engineering*,

BIBLIOGRAPHY

- vol. 19, no. 11, pp. 1028–1044, Nov. 1993, ISSN: 0098-5589. DOI: 10.1109/32.256851.
- [16] M. Cadariu, E. Bouwers, J. Visser and A. van Deursen, ‘Tracking known security vulnerabilities in proprietary software systems’, in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, Montreal, QC, Canada, Mar. 2015, pp. 516–519, ISBN: 978-1-4799-8469-5. DOI: 10.1109/SANER.2015.7081868.
- [17] M. Castro, M. Costa and T. Harris, ‘Securing software by enforcing data-flow integrity’, in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI ’06, Seattle, Washington: USENIX Association, 6th Nov. 2006, pp. 147–160, ISBN: 1-931971-47-1.
- [18] S. Chacon and B. Straub, *Pro git*, 2nd ed. Apress, 2014, ISBN: 978-1-4842-0076-6. DOI: 10.1007/978-1-4842-0076-6.
- [19] *Common vulnerability fix*. [Online]. Available: <https://github.com/UKGovLD/registry-config-base/commit/4122b272> (visited on 28/11/2018).
- [20] C. Cowan, ‘Software security for open-source systems’, *IEEE Security & Privacy*, vol. 99, no. 1, pp. 38–45, 19th Feb. 2003, ISSN: 1540-7993. DOI: 10.1109/MSECP.2003.1176994.
- [21] K. Crowston, K. Wei, J. Howison and A. Wiggins, ‘Free/libre open-source software development: What we know and what we do not know’, *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, p. 7, 1st Feb. 2012, ISSN: 0360-0300. DOI: 10.1145/2089125.2089127.
- [22] *Curl repository*. [Online]. Available: <https://github.com/curl/curl> (visited on 31/10/2018).
- [23] *Cve – about cwe*, Mitre Corporation, 30th Mar. 2018. [Online]. Available: <https://cwe.mitre.org/about/index.html> (visited on 09/10/2018).

BIBLIOGRAPHY

- [24] *Cve – frequently asked question (faq)*, Mitre Corporation, 30th Mar. 2018. [Online]. Available: <https://cwe.mitre.org/about/faq.html#A.1> (visited on 09/10/2018).
- [25] *Cve – home*, Mitre Corporation, 17th Jan. 2018. [Online]. Available: <https://cve.mitre.org/about/index.html> (visited on 09/10/2018).
- [26] *Cve-2019-0211*. [Online]. Available: <https://people.canonical.com/~ubuntu-security/cve/2019/CVE-2019-0211.html> (visited on 03/04/2019).
- [27] *Cve-identified vulnerability fix*. [Online]. Available: <https://github.com/Constantiner/resolve-node-configs-hierarchy/commit/5f3ad> (visited on 28/11/2018).
- [28] L. Dabbish, C. Stuart, J. Tsay and J. Herbsleb, ‘Social coding in github: Transparency and collaboration in an open software repository’, in *Proceedings of the ACM 2012 conference on computer supported cooperative work*, ACM, 11th Feb. 2012, pp. 1277–1286. DOI: 10.1145/2145204.2145396.
- [29] S. Dashevskyi, A. D. Brucker and F. Massacci, ‘A screening test for disclosed vulnerabilities in foss components’, *IEEE Transactions on Software Engineering*, 2018, ISSN: 0098-5589. DOI: 10.1109/TSE.2018.2816033.
- [30] D. Evans and D. Larochelle, ‘Improving security using extensible light-weight static analysis’, *IEEE Software*, vol. 19, no. 1, pp. 42–51, Jan. 2002, ISSN: 0740-7459. DOI: 10.1109/52.976940.
- [31] *Extensible markup language (xml)*. [Online]. Available: <https://www.w3.org/XML> (visited on 14/10/2018).
- [32] M. Fagan, M. M. H. Khan and R. Buck, ‘A study of users’ experiences and beliefs about software update messages’, *Computers in Human Behavior*, vol. 51, pp. 504–519, Oct. 2015, ISSN: 0747-5632. DOI: 10.1016/j.chb.2015.04.075.
- [33] *Git commit code difference*. [Online]. Available: <https://github.com/doriandekoning/Contextproject-BeNine/commit/a8b4eb> (visited on 28/11/2018).

BIBLIOGRAPHY

- [34] *Github*, GitHub. [Online]. Available: <https://github.com> (visited on 25/10/2018).
- [35] *Gitpython*. [Online]. Available: <https://github.com/gitpython-developers/GitPython> (visited on 20/09/2018).
- [36] *Google code archive*. [Online]. Available: <https://code.google.com/archive> (visited on 25/10/2018).
- [37] A. hArora, A. Nandkumar and R. Telang, ‘Does information security attack frequency increase with vulnerability disclosure? an empirical analysis’, *Information Systems Frontiers*, vol. 8, no. 5, pp. 350–362, 1st Dec. 2006, ISSN: 1572-9419. DOI: 10.1007/s10796-006-9012-5.
- [38] A. E. Hassan, ‘The road ahead for mining software repositories’, in *2008 Frontiers of Software Maintenance*, IEEE, Sep. 2008, pp. 48–57. DOI: 10.1109/FOSM.2008.4659248.
- [39] J.-H. Hoepman and B. Jacobs, ‘Increased security through open source’, *Communications of the ACM*, vol. 50, no. 1, pp. 79–83, 1st Jan. 2007, ISSN: 0001-0782. DOI: 10.1145/1188913.1188921.
- [40] M. Jimenez, M. Papadakis and Y. L. Traon, ‘An empirical analysis of vulnerabilities in openssl and the linux kernel’, in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, Dec. 2016, pp. 105–112, ISBN: 978-1-5090-5575-3. DOI: 10.1109/APSEC.2016.025.
- [41] *Json*, ecma International. [Online]. Available: <https://www.json.org> (visited on 13/10/2018).
- [42] *Json - json encoder and decoder*, Python Software Foundation. [Online]. Available: <https://docs.python.org/3/library/json.html> (visited on 20/02/2019).
- [43] H. Kagdi, M. L. Collard and J. I. Maletic, ‘A survey and taxonomy of approaches for mining software repositories in the context of software evolution’, *Journal of software maintenance and evolution: Research and practice*, vol. 19, no. 2, pp. 77–131, 29th Mar. 2007. DOI: 10.1002/smr.344.

BIBLIOGRAPHY

- [44] D. Kawrykow and M. P. Robillard, ‘Non-essential changes in version histories’, in *2011 33rd International Conference on Software Engineering (ICSE)*, IEEE, Honolulu, HI, USA, 2011-10-10, pp. 351–360, ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985842.
- [45] K. Khan, J. Han and Y. Zheng, ‘A framework for an active interface to characterise compositional security contracts of software components’, in *Proceedings 2001 Australian Software Engineering Conference*, Canberra, ACT, Australia, Australia: IEEE, 2001, pp. 117–126, ISBN: 0-7695-1254-2. DOI: 10.1109/ASWEC.2001.948505.
- [46] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi and J. Hu, ‘Vulpecker: An automated vulnerability detection system based on code similarity analysis’, in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC ’16, Los Angeles, California, USA: ACM, 2016, pp. 201–213, ISBN: 978-1-4503-4771-6. DOI: 10.1145/2991079.2991102.
- [47] M. Linares-Vásquez, G. Bavota and C. Escobar-Velásquez, ‘An empirical study on android-related vulnerabilities’, in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, IEEE, 21st May 2017, pp. 2–13, ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.60.
- [48] *Linux cve fix*. [Online]. Available: <https://github.com/torvalds/linux/commit/cc255c76> (visited on 09/11/2018).
- [49] *Linux kernel source tree*. [Online]. Available: <https://github.com/torvalds/linux> (visited on 20/09/2018).
- [50] B. Livshits and M. Lam, ‘Finding security vulnerabilities in java applications with static analysis.’, in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM’05, vol. 14, USENIX Association, 2005, pp. 271–286.
- [51] B. Livshits and T. Zimmermann, ‘Dynamine: Finding common error patterns by mining software revision histories’, in *ACM SIGSOFT Software Engineering Notes*, vol. 30, ACM, Sep. 2005, pp. 296–305, ISBN: ISBN:1-59593-014-0. DOI: 10.1145/1095430.1081754.

BIBLIOGRAPHY

- [52] E. Marcussen, *Grep rough audit - source code auditing tool*. [Online]. Available: <https://github.com/wireghoul/graudit> (visited on 23/02/2019).
- [53] F. Massacci and V. H. Nguyen, ‘Which is the right source for vulnerability studies?: An empirical analysis on mozilla firefox’, in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, ser. MetriSec ’10, Bolzano, Italy: ACM, 15th Sep. 2010, p. 4, ISBN: 978-1-4503-0340-8. DOI: 10.1145/1853919.1853925.
- [54] A. Mathur and M. Chetty, ‘Impact of user characteristics on attitudes towards automatic mobile application updates’, in *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, USENIX Association, 2017, pp. 175–193.
- [55] M. Matsushita, K. Sasaki and K. Inoue, ‘Coxr: Open source development history search system’, in *12th Asia-Pacific Software Engineering Conference (APSEC’05)*, IEEE, Dec. 2005. DOI: 10.1109/APSEC.2005.56.
- [56] A. Meneely, H. Srinivasan, A. Musa, A. Tejeda, M. Mokary and B. Spates, ‘When a patch goes bad: Exploring the properties of vulnerability contributing commits’, in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, IEEE, Oct. 2013, pp. 65–74, ISBN: 978-0-7695-5056-5. DOI: 10.1109/ESEM.2013.19.
- [57] A. Mockus and L. G. Votta, ‘Identifying reasons for software changes using historic databases’, in *Proceedings 2000 International Conference on Software Maintenance*, IEEE, 2000, pp. 120–130, ISBN: 0-7695-0753-0. DOI: 10.1109/ICSM.2000.883028.
- [58] R. Moser, W. Pedrycz and G. Succi, ‘A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction’, in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE ’08, Leipzig, Germany: ACM, 2008, pp. 181–190, ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368114.

BIBLIOGRAPHY

- [59] *Multiprocessing - process-based parallelism*, Python Software Foundation. [Online]. Available: <https://docs.python.org/3.7/library/multiprocessing.html> (visited on 23/04/2019).
- [60] J. C. Munson and T. M. Khoshgoftaar, ‘The detection of fault-prone programs’, *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 423–433, May 1992, ISSN: 0098-5589. DOI: 10.1109/32.135775.
- [61] S. Neuhaus and B. Plattner, ‘Software security economics: Theory, in practice’, in *The Economics of Information Security and Privacy*, Springer, 8th Oct. 2013, pp. 75–92, ISBN: 978-3-642-39498-0. DOI: 10.1007/978-3-642-39498-0_4.
- [62] V. H. Nguyen and L. M. S. Tran, ‘Predicting vulnerable software components with dependency graphs’, in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, ser. MetriSec ’10, ACM, 2010, p. 3, ISBN: 978-1-4503-0340-8. DOI: 10.1145/1853919.1853923.
- [63] N. Nurseitov, M. Paulson, R. Reynolds and C. Izurieta, ‘Comparison of json and xml data interchange formats: A case study’, vol. 9, Jan. 2009, pp. 157–162.
- [64] *Nvd - home*. [Online]. Available: <https://nvd.nist.gov> (visited on 29/10/2018).
- [65] *Oauth2 proxy csrf fix*. [Online]. Available: https://github.com/bitly/oauth2_proxy/commit/44646 (visited on 09/11/2018).
- [66] *Openssl tls/ssl and crypto library*, OpenSSL. [Online]. Available: <https://github.com/openssl/openssl> (visited on 31/10/2018).
- [67] T. J. Ostrand and E. J. Weyuker, ‘A tool for mining defect-tracking systems to predict fault-prone files’, in *Proc. of Int. Workshop on Mining Software Repositories*, Institution of Engineering and Technology, Jan. 2004, pp. 85–89. DOI: 10.1049/ic:20040482.

BIBLIOGRAPHY

- [68] L. B. Othmane, G. Chehrazi, E. Bodden, P. Tsalovski, A. D. Brucker and P. Miseldine, ‘Factors impacting the effort required to fix security vulnerabilities’, in *Information Security*, J. Lopez and C. Mitchell, Eds., Cham: Springer International Publishing, 2015, pp. 102–119, ISBN: 978-3-319-23318-5. DOI: 10.1007/978-3-319-23318-5_6.
- [69] *Owasp dependency check*, The Open Web Application Security Project (OWASP), 16th Sep. 2018. [Online]. Available: https://www.owasp.org/index.php/OWASP_Dependency_Check (visited on 06/10/2018).
- [70] *Owasp home*, The Open Web Application Security Project (OWASP), 18th Sep. 2018. [Online]. Available: https://www.owasp.org/index.php/Main_Page (visited on 29/09/2018).
- [71] *Owasp top ten 2004 project*, The Open Web Application Security Project (OWASP), 27th Jan. 2004. [Online]. Available: https://www.owasp.org/index.php/Top_10_2004 (visited on 21/04/2019).
- [72] *Owasp top ten 2017 project*, The Open Web Application Security Project (OWASP), 20th Oct. 2017. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project (visited on 26/09/2018).
- [73] C. Payne, ‘On the security of open source software’, *Information Systems Journal*, vol. 12, no. 1, pp. 61–78, 8th Feb. 2002, ISSN: 1350-1917. DOI: 10.1046/j.1365-2575.2002.00118.x.
- [74] *Pep 8 – style guide for python code*, Python Software Foundation. [Online]. Available: <https://www.python.org/dev/peps/pep-0008> (visited on 15/02/2019).
- [75] *Performance tips*, Python Software Foundation. [Online]. Available: <https://wiki.python.org/moin/PythonSpeed/PerformanceTips> (visited on 15/02/2019).
- [76] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl and Y. Acar, ‘Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits’, in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Secur-*

BIBLIOGRAPHY

- ity*, ser. CCS '15, Denver, Colorado, USA: ACM, 2015, pp. 426–437, ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813604.
- [77] W. Poncin, A. Serebrenik and M. V. D. Brand, ‘Process mining software repositories’, in *2011 15th European Conference on Software Maintenance and Reengineering*, Oldenburg, Germany: IEEE, 2011, pp. 5–14, ISBN: 978-1-61284-259-2. DOI: 10.1109/CSMR.2011.5.
- [78] *Progpilot*, Design security. [Online]. Available: <https://github.com/designsecurity/progpilot> (visited on 15/04/2019).
- [79] *Pyqt5*, Riverbank Computing Limited. [Online]. Available: <https://www.riverbankcomputing.com/software/pyqt/intro> (visited on 13/04/2019).
- [80] *Python programming language repository*. [Online]. Available: <https://github.com/python/cpython> (visited on 31/10/2018).
- [81] *Re - regular expression operations*, Python Software Foundation. [Online]. Available: <https://docs.python.org/3/library/re.html> (visited on 20/02/2019).
- [82] S. Reis and R. Abreu, *Secbench mining tool*, The Quasar Research Group. [Online]. Available: <https://github.com/TQRG/secbench-mining-tool> (visited on 11/10/2018).
- [83] S. Reis and R. Abreu, ‘Secbench: A database of real security vulnerabilities’, *Secure Software Engineering in DevOps and Agile Development*, M. G. Jaatun and D. S. Cruzes, Eds., pp. 69–85, 31st Oct. 2017.
- [84] *Retire.js*. [Online]. Available: <https://retirejs.github.io/retire.js> (visited on 11/03/2018).
- [85] G. Robles, ‘Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings’, in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*, IEEE, May 2010, pp. 171–180, ISBN: 978-1-4244-6803-4. DOI: 10.1109/MSR.2010.5463348.

BIBLIOGRAPHY

- [86] R. L. Russell, L. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood and M. W. McConley, ‘Automated vulnerability detection in source code using deep representation learning’, pp. 757–762, 11th Jul. 2018. DOI: 10.1109/ICMLA.2018.00120.
- [87] *Safety - python dependency checker*, pyup. [Online]. Available: <https://github.com/pyupio/safety> (visited on 03/11/2018).
- [88] R. Scandariato, J. Walden, A. Hovsepyan and W. Joosen, ‘Predicting vulnerable software components via text mining’, *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 1st Oct. 2014, ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2340398.
- [89] G. Schryen, ‘Is open source security a myth?’, *Communications of the ACM*, vol. 54, no. 5, pp. 130–140, 1st May 2011, ISSN: 0001-0782. DOI: 10.1145/1941487.1941516.
- [90] R. W. Selby and A. A. Porter, ‘Learning from examples: Generation and evaluation of decision trees for software resource analysis’, *IEEE Transactions on Software Engineering*, vol. 14, no. 12, pp. 1743–1757, Dec. 1988, ISSN: 0098-5589. DOI: 10.1109/32.9061.
- [91] W. Shang, Z. M. Jiang, B. Adams and A. E. Hassan, ‘Mapreduce as a general framework to support research in mining software repositories (msr)’, in *2009 6th IEEE International Working Conference on Mining Software Repositories*, IEEE, May 2009, pp. 21–30, ISBN: 978-1-4244-3493-0. DOI: 10.1109/MSR.2009.5069477.
- [92] J. Śliwerski, T. Zimmermann and A. Zeller, ‘When do changes induce fixes?’, *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005. DOI: 10.1145/1082983.1083147.
- [93] *Sourceforge - download, develop and publish free open source software*, Slashdot Media. [Online]. Available: <https://sourceforge.net> (visited on 25/10/2018).
- [94] D. Spadini, M. Aniche and A. Bacchelli, *Pydriller*. [Online]. Available: <https://github.com/ishepard/pydriller> (visited on 15/02/2018).

BIBLIOGRAPHY

- [95] D. Spadini, M. Aniche and A. Bacchelli, ‘Pydriller: Python framework for mining software repositories’, in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ES-EC/FSE 2018, ACM, 2018, pp. 908–911, ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3264598.
- [96] *Spotbugs*. [Online]. Available: <https://spotbugs.github.io> (visited on 15/04/2019).
- [97] *Tkinter - python interface to tcl/tk*, Python Software Foundation. [Online]. Available: <https://docs.python.org/3/library/tkinter.html> (visited on 13/04/2019).
- [98] *Unittest - unit testing framework*, Python Software Foundation. [Online]. Available: <https://docs.python.org/3/library/unittest.html> (visited on 14/10/2018).
- [99] J. Walden, J. Stuckman and R. Scandariato, ‘Predicting vulnerable components: Software metrics vs text mining’, in *2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 3rd Nov. 2014, pp. 23–33. DOI: 10.1109/ISSRE.2014.32.
- [100] *Welcome to python.org*, Python Software Foundation. [Online]. Available: <https://www.python.org> (visited on 13/10/2018).
- [101] D. A. Wheeler, *Flawfinder: A static analysis tool for finding vulnerabilities in c/c++ source code*. [Online]. Available: <https://github.com/david-a-wheeler/flawfinder> (visited on 15/02/2019).
- [102] C. C. Williams and J. K. Hollingsworth, ‘Automatic mining of source code repositories to improve bug finding techniques’, *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466–480, Jun. 2005, ISSN: 0098-5589. DOI: 10.1109/TSE.2005.63.
- [103] B. Witten, C. Landwehr and M. Caloyannides, ‘Does open source improve system security?’, *IEEE Software*, vol. 18, no. 5, pp. 57–61, Sep. 2001, ISSN: 0740-7459. DOI: 10.1109/52.951496.

BIBLIOGRAPHY

- [104] M. Zitser, R. Lippmann and T. Leek, ‘Testing static analysis tools using exploitable buffer overflows from open source code’, in *ACM SIGSOFT Software Engineering Notes*, vol. 29, ACM, 2004, pp. 97–106, ISBN: 1-58113-855-5. DOI: 10.1145/1029894.1029911.

Appendices

A Performance Testing Results

Repository	Total commits	Time (hh:mm:ss)
C/C++		
curl	24,173	0:38:37
httpd	31,399	1:16:21
icu	30,642	9:03:57
ImageMagick	15,503	1:12:41
libarchive	5,461	0:13:15
libpng	4,054	0:36:14
libtomcrypt	1,906	0:04:01
libxml2	4,737	0:32:35
libxslt	1,776	0:03:40
linux	825,898	73:54:55
openssl	23,714	0:54:28
v8	55,289	9:38:54
xalan-c	4,323	0:07:38
xerces-c	6,358	0:16:11
zlib	419	0:01:29
Java		
activemq	10,158	0:32:46
axis2-java	13,329	1:05:14
batik	3,490	0:14:28
bc-java	5,530	0:25:21
camel	36,562	4:49:29

APPENDICES

cocoon	13,156	0:25:47
commons-fileupload	950	0:00:38
cordova-android	3,760	0:05:04
cxfr	14,879	1:24:13
flex-sdk	33,707	1:22:20
geronimo	13,137	0:27:42
httpcomponents-client	2,993	0:09:33
jetty.project	16,621	1:00:20
poi	9,786	0:38:15
spring-framework	18,415	0:56:35
spring-security	7,616	0:12:14
struts	5,649	0:09:16
tomcat	20,723	0:54:59
wss4j	2,543	0:12:30
xalan-j	4,560	0:10:17
Python		
ansible	44,388	4:39:53
asn1crypto	586	0:09:38
bcrypt	165	0:00:25
botocore	5,666	0:55:11
cpython	103,752	17:01:44
cryptography	7,619	1:09:44
django	26,822	5:35:03
flask	3,505	0:14:14
home-assistant	18,675	1:56:53
paramiko	3,231	0:13:29
photon	3,433	0:04:22
pyopenssl	1,999	0:33:27
python-certifi	140	0:00:03
requests	5,614	0:35:14
sqlmap	8,671	0:42:55
tornado	4,054	0:26:48
urllib3	3,190	0:20:28

APPENDICES

Total (52 repositories)	1,514,726	148:31:28
--------------------------------	------------------	------------------

Table A1: The time taken to complete the analysis process of each repository.

B RegExp match, all repositories

Vulnerabilities	Total	Percentage
Broken Access Control	190	0.13%
Broken Authentication and Session Management	978	0.67%
Buffer Overflow	1,365	0.94%
Bug Tracker Issue	34,799	23.90%
Context Leaks	17	0.01%
Cross-Site Request Forgery	204	0.14%
Cross-Site Scripting	310	0.21%
Distributed Denial-of-Service / Denial-of-Service	7,340	5.04%
Encryption Issues	24,291	16.68%
Hard Coded	1,766	1.21%
Injection	1,772	1.22%
Insufficient Attack Protection	191	0.13%
Memory Leaks	5,819	4.00%
Miscellaneous	14,409	9.89%
Null Pointers	7,803	5.36%
Overflow	4,929	3.38%
Resource Leaks	98	0.07%
Path / Directory Traversal	177	0.12%
SHA-1 Collision	1	0.00%
Security Misconfiguration	10,457	7.18%
Sensitive Data Exposure	22,657	15.56%
Using Components with Known Vulnerabilities	186	0.13%
Underprotected APIs	5,862	4.03%
Total	145,621	100.00%

APPENDICES

Table A2: All vulnerabilities matched by the regular expressions in all repositories analysed.

C RegExp match, C/C++ repositories

Vulnerabilities	Total	Percentage
Broken Access Control	142	0.12%
Broken Authentication and Session Management	545	0.47%
Buffer Overflow	1,286	1.10%
Bug Tracker Issue	29,884	25.62%
Context Leaks	14	0.01%
Cross-Site Request Forgery	15	0.01%
Cross-Site Scripting	97	0.08%
Distributed Denial-of-Service / Denial-of-Service	6,129	5.25%
Encryption Issues	17,022	14.59%
Hard Coded	1,449	1.24%
Injection	1,413	1.21%
Insufficient Attack Protection	172	0.15%
Memory Leaks	5,242	4.49%
Miscellaneous	12,741	10.92%
Null Pointers	7,397	6.34%
Overflow	4,486	3.85%
Resource Leaks	69	0.06%
Path / Directory Traversal	159	0.14%
SHA-1 Collision	1	0.00%
Security Misconfiguration	6,314	5.41%
Sensitive Data Exposure	17,448	14.96%
Using Components with Known Vulnerabilities	167	0.14%
Underprotected APIs	4,463	3.83%
Total	116,655	100.00%

Table A3: All vulnerabilities matched by the regular expressions in the C/C++ repositories analysed.

D RegExp match, Java repositories

Vulnerabilities	Total	Percentage
Broken Access Control	21	0.12%
Broken Authentication and Session Management	360	2.03%
Buffer Overflow	27	0.15%
Bug Tracker Issue	4,304	24.25%
Context Leaks	1	0.01%
Cross-Site Request Forgery	72	0.41%
Cross-Site Scripting	198	1.12%
Distributed Denial-of-Service / Denial-of-Service	849	4.78%
Encryption Issues	3,339	18.82%
Hard Coded	210	1.18%
Injection	203	1.14%
Insufficient Attack Protection	8	0.05%
Memory Leaks	242	1.36%
Miscellaneous	640	3.61%
Null Pointers	326	1.84%
Overflow	85	0.48%
Resource Leaks	13	0.07%
Path / Directory Traversal	7	0.04%
Security Misconfiguration	3,468	19.54%
Sensitive Data Exposure	2,674	15.07%
Using Components with Known Vulnerabilities	3	0.02%
Underprotected APIs	695	3.92%
Total	17,745	100.00%

Table A4: All vulnerabilities matched by the regular expressions in the Java repositories analysed.

E RegExp match, Python repositories

Vulnerabilities	Total	Percentage
Broken Access Control	27	0.24%
Broken Authentication and Session Management	73	0.65%

APPENDICES

Buffer Overflow	52	0.46%
Bug Tracker Issue	611	5.45%
Context Leaks	2	0.02%
Cross-Site Request Forgery	117	1.04%
Cross-Site Scripting	15	0.13%
Distributed Denial-of-Service / Denial-of-Service	362	3.23%
Encryption Issues	3,930	35.02%
Hard Coded	107	0.95%
Injection	156	1.39%
Insufficient Attack Protection	11	0.10%
Memory Leaks	335	2.99%
Miscellaneous	1,028	9.16%
Null Pointers	80	0.71%
Overflow	358	3.19%
Resource Leaks	16	0.14%
Path / Directory Traversal	11	0.10%
Security Misconfiguration	675	6.02%
Sensitive Data Exposure	2,535	22.59%
Using Components with Known Vulnerabilities	16	0.14%
Underprotected APIs	704	6.27%
Total	11,221	100.00%

Table A5: All vulnerabilities matched by the regular expressions in the Python repositories analysed.