The University Of Sheffield.

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 Enginnering by Zer Jun Eng

COM3610

5th November 2018

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 : 5th November 2018

Abstract

Each time a vulnerability is identified in a Free/Libre and Open Source Software (FLOSS) project, the developers will start the fixing process and publish several commits to address the vulnerability. The same condition also applies to FLOSS components, which are widely used in both proprietary and open source softwares. The purpose of this project is to develop a repository mining tool that is able to detect commits that fix both known and unknown vulnerabilities. From the identified commits, the tool can determine whether an application is using a vulnerable component.

To improve the correctness of the mining results, the tool can be extended to evaluate the lines of code changed between commits to identify the real vulnerability fixing commits. *RESULTS OBTAINED ARE BRIEFLY DISCUSSED HERE*

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

List of Tables

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, modify and redistribute the software's source code [12]. Since the introduction of the version control system, many repository hosting sites such as SourceForge [56], Google Code [21], and GitHub [18] 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 softwares quality, which included fixing software vulnerabilities [16].

Building a 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 softwares [29]. An important factor of the 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**) [38] 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 clearly record 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 vulnerable methods. This information is often described explicitly in the vulnerability report such as the NVD and the CVE. Therefore, identifying the vulnerability fixing commits is a great approach of locating the vulnerable lines of code, which allows checking 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 [5]. 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 most popular security issues in OWASP Top Ten Project. The patterns should be expressed using regular expressions.
- 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 a suitable file format such as JSON, XML, or CSV.

- Extend the mining tool which checks the code difference in the commits found to obtain the actual commits fixing the security vulnerabilities. This extension should separate from the mining process to make the mining results easier to verify and debug.

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 780,000 commits in total [30] 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 process 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 softwares 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 softwares. 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 softwares 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 [24, 60], and provides more flexibility and freedom over the security option to their users [45]. 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 [53].

Arora, Nandkumar and Telang [22] 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 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 [13]. 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. [14]. 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.

2.3 Common Vulnerabilities and Exposures

The Common Vulnerabilities and Exposures (**CVE**) is another security project launched by the Mitre Corporation [15] 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 in 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.

2.4 Security Issues in Open Source Softwares

The Open Web Application Security Project (**OWASP**) is a worldwide non-profit organization committed to improve and raise the awareness of software security [43]. 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 [44], one of the vulnerabilities that is closely related to this project is *Using Components with Known Vulnerabilities*, which will be extensively discussed.

2.4.1 Using Components with Known Vulnerabilities

It has been indicated that a small software component could create a large error in a software system [7, 35, 54]. Components such as plugins, libraries, and modules are ubiquitous in both open source and proprietary softwares. Third-party components are increasingly being integrated into softwares to reduce the amount of time and effort required for development [6], but they also increase the risk of vulnerabilities being introduced into the softwares. 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 focuses on studying 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 [42], Retire.js [49], and Safety [51] are applications that identifies vulnerable dependencies in a software project. Cadariu et al. [10] have used the OWASP Dependency Check tool to find all known vulnerabilities that have a unique CVE identifier in proprietary softwares 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. 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 [9] 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. [52] 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 softwares. 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.

2.5 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 [46]. The purpose of MSR is to extract practical information from rich metadata and discover hidden trends about a specific evolutionary characteristic [28]. 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 [23]. 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 [50]. 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 [55] suggested that the MSR process should be split into several stages, with each stage focusing on a specific topic of the problem to achieve the optimal efficiency.

2.5.1 Keywords Search

Keywords search is the core procedure of retrieving information from a repository. For many complex approaches, the 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 from a query, it is required to integrate some algorithms and modules into the search function. Matsushita, Sasaki, and Inoue [31] 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 into every commit. On the contrary, Mockus and Votta [33] 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.

2.5.2 Finding Vulnerabilities

Meneely et al. [32] conducted a research to study the properties of commits that introduce vulnerabilities, and it is found that most vulnerabilities required an average 1175 days to fix. For every vulnerability identified in a repository, the vulnerability fixing process that involves analysis, implementation, testing, and release will be executed [41]. Most of the vulnerabilities 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 repository mining tool to determine the correctness of the commit. In this situation, it is required to investigate the changed files to study the lines of code modified. This is a demanding task and often required manual investigation as code changes between commits are highly variable.

Cowan [11] suggested a list of software auditing and vulnerability mitigation tools that perform static code and runtime analysis to find bugs and vulnerabilities. This approach is popular because it is able to find simple bugs in a short amount of time. However, Bessey et al. [8] claimed that static tools have a negative effect on technical development due to its high false positives nature. 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 [34]. It might require several experiments of different configurations to obtain the best result, and the result may vary across different data sets.

This project extends prior work on Reis and Abreu's [47] Secbench Mining Tool. 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 [48] 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, it is not practical to perform manual evaluation on every result. In this project, the tool developed should be able to automate the evaluation process to some extent, while preserving the accuracy of the results.

2.5.3 Vulnerability Patch Prediction

Using the results retrieved from the searching procedure, it is able to predict whether a certain commit message is indicating a vulnerability patch. Williams and Hollingsworth [59] have developed a source code analysis tool that searches for bug fixes and combines with information mined from repositories to improve the results. It is stated that the most efficient way to utilise the historical information is to ignore the commit messages and focus on mining the code changes. In order to locate the actual code changes for the bug fix, a function return value checker was implemented to compares the number of warnings produced by the same function across different versions. Williams and Hollingsworth assumed that a bug is fixed if the warnings produced by the same function have decreased between two versions, and the final result produced is a list of functions that are related to a potential bug fix in the commit history. However, Ostrand and Weyuker [40] focused on predicting the bug fixes through the most frequently modified files between version releases. It is hypothesised that a bug is more likely to occur in files that have more lines of code, as this implies that the files are more important and contain more functionality than others. Although these techniques are not focusing on predicting vulnerability patches, they can be reused and adjusted to suit this project needs.

Chapter 3

Requirements and Analysis

The purpose of this chapter is to express the aims in more 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 a time-consuming process. Additionally, it would 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 relative commits. It will be 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 will be discussed in the later section.

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. However, all previous related work performed did not use automated techniques for the evaluation. 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 Specification

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
Repeatable: The results should be repeatable and reproducible.	Essential
Robustness: The mining tool should be able to handle all possible errors without terminating the mining process.	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

Table 3.1: Specification of the mining tool

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 [58] 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

Since the mining tool is decided to be programmed in Python 3, a wide range of libraries could be integrated to enhance its functionality.

- PyGithub is a Python library build to access the GitHub API [26].
- GitPython is a Python library build to interact with Git repositories using a combination of python and git command implementation [20].

3.3.3 File Format of Result

The JavaScript Object Notation (**JSON**) [27] 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**) [17] 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 [37], 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: the **mining** process and the **evaluation** process. The **mining** process takes a Git repository as input, searches through the commit log, and stores the list of commits that might potentially contain a patch in a JSON file. The **evaluation** process takes a JSON file as input, and check the code difference of every commit in the log file to identify the real patches.

3.5 Problems and Constraints

As mentioned in **Section 1.3**, the main challenges of this project are **data**, **misclassification**, **evaluation** and **time**. The subsequent challenge is the implementation difficulties, which the severity is dependent on the complexity of the problems and the resources available. It is also expected that some problems might not be solved and new problems could emerge in the course of the project. This section will discuss the problems in detail and review several ways of mitigating them, as well as analysing the possible constraints that might affect the progress of the project.

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. 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.

There are several ways of mitigating the problems to reduce the risk, provided that the problems are clearly identified and they are under the project scope. It is estimated that the **evaluation** process would be the biggest challenge of this project since it was regarded as a complicated and difficult area in previous researches. Moreover, this project plans to implement an automated version of the evaluation process, which will further increase the difficulty level. The implementation of automated evaluation is hard and it does not guarantee to provide a good result. It is also extremely challenging for the mining tool to work across repositories programmed in different programming languages. The constraint is that the tool has to be exhaustively tested to find the optimal threshold value and for it to be automated and produce good results. Although the tool might produce good results on some repositories, it does not indicate that the tool will produce consistent results on all repositories. To ensure the minimum quality of the results, one of the solutions might be using both automated method for basic filtering and a manual method for advance refinement.

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 [57], 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 #	Test Data	Expected Result	Actual Result	Status

Table 3.2: Documentation format of the unit testing

3.6.2 System Testing

After completing the unit testing, the mining tool has to be tested for its completeness and robustness, as mentioned in **Table 3.1**. It is expected that the program would not be able to handle complicated errors during the early implementation, and the project schedule would become an iterative process between implementation and testing. It is assumed that the testing stage would be the most time-consuming process in the whole project, thus it might be required to allocate more time and effort into this stage.

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. Evaluating the mining tool on several real-world projects will test its ability of handling 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 [30], Apache HTTP Server [2], Apache Tomcat [3], GitLab Community Edition [19], Homebrew core [25], Nixpkgs [36] and Odoo [39] are a good starting point for this project as they all have a large number of commits. This approach is reasoable 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.
- **Efficiency**: The total time taken required for the tool to complete the seaching process.

3.8 Ethical Issues

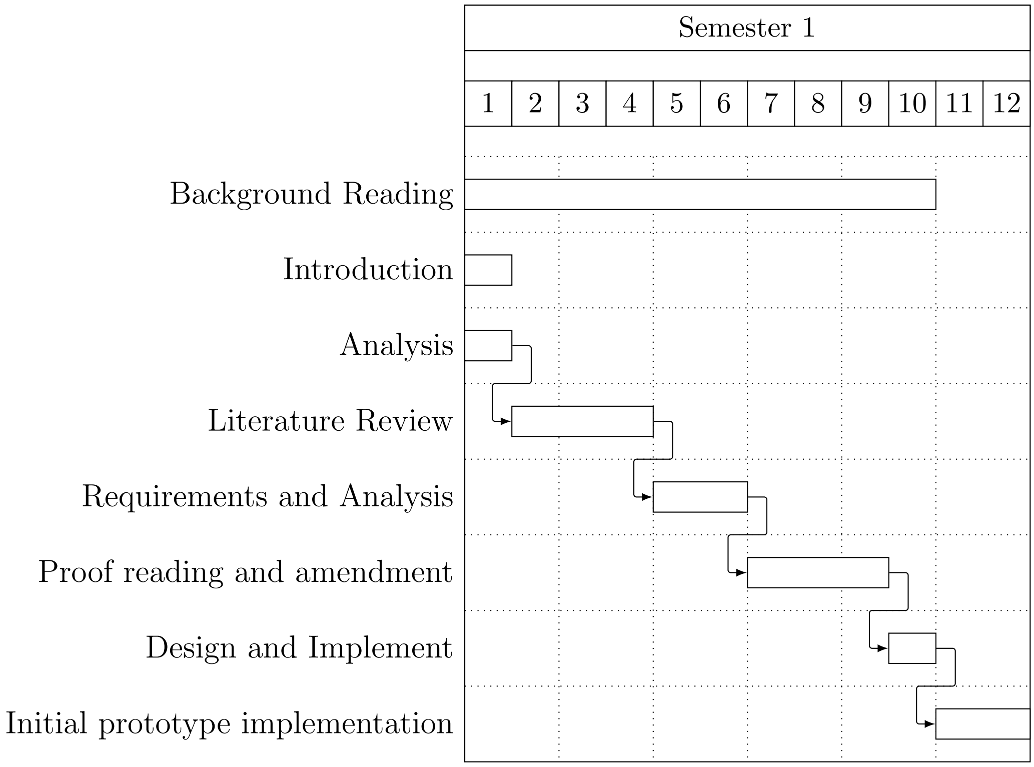
In this project, it has to be clearly declared that any known or unknown vulnerabilities found by the mining tool in any repositories will not be publicly disclosed without the permission of the original authors. The reason is that publishing the vulnerabilities publicly would make the softwares highly vulnerable to attackers [4], and it is recommended to wait for the official announcement from the software vendors.

Chapter 4

Conclusions and Project Plan

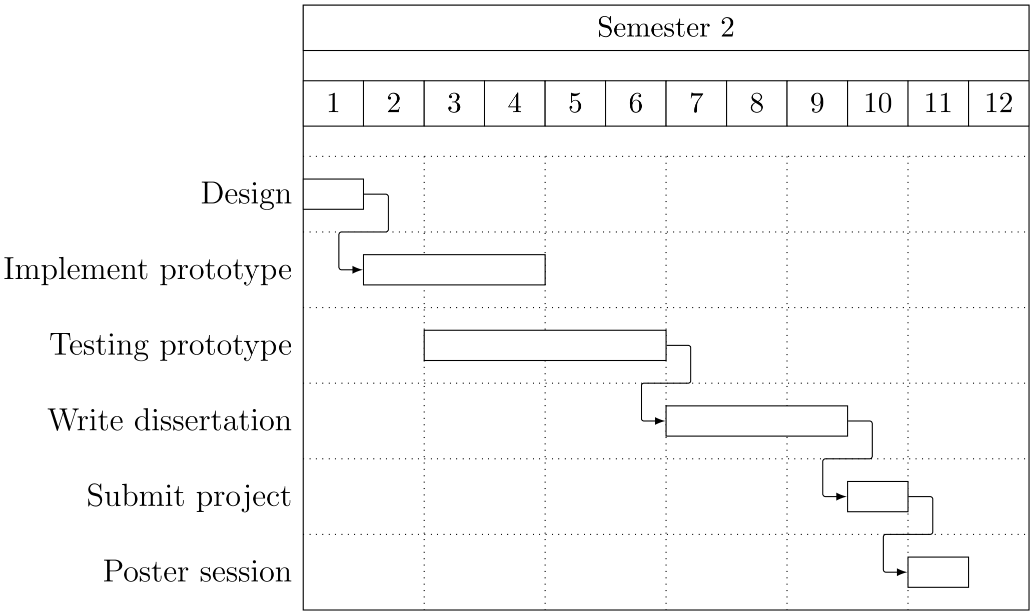
4.1 Plan of Action

Semester 1 and Christmas Vacation



- **Week 7**: Starting from this week, discuss with the supervisor weekly about the document, also it is best to start the design stage early, and show the prototype to the supervisor.
- **Week 11**: If the design stage is started early, then it is hoped to produce the initial prototype of the tool before Christmas Vacation.
- **Christmas Vacation**: Regularly work on the implementation of the tool and push the commits.

4.1.1 Semester 2



- **Week 1**: Finished the design and started implementation during holiday.
- **Week 2**: The implementing and testing stage is a repetitive process. It is very likely that the program will run into errors in the testing and had to spend more time fixing it.

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] *Apache http server repository.* [Online]. Available: `https://github.com/apache/httpd` (visited on 31/10/2018).

[3] *Apache tomcat repository.* [Online]. Available: `https://github.com/apache/tomcat` (visited on 31/10/2018).

[4] 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`.

[5] 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`.

[6] 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`.

[7] 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`.

[8] 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`.

[9] 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*, vol. 19, no. 11, pp. 1028–1044, Nov. 1993, ISSN: 0098-5589. DOI: `10.1109/32.256851`.

[10] 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`.

[11] 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`.

[12] 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`.

[13] *Cve – about cwe*, Mitre Corporation, 30th Mar. 2018. [Online]. Available: `https://cwe.mitre.org/about/index.html` (visited on 09/10/2018).

[14] *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).

[15] *Cve – home*, Mitre Corporation, 17th Jan. 2018. [Online]. Available: `https://cve.mitre.org/about/index.html` (visited on 09/10/2018).

[16] 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.

[17] *Extensible markup language (xml).* [Online]. Available: https://www.w3.org/XML/ (visited on 14/10/2018).

[18] *Github*, GitHub. [Online]. Available: https://github.com/ (visited on 25/10/2018).

[19] *Gitlab community edition repository.* [Online]. Available: https://github.com/gitlabhq/gitlabhq (visited on 31/10/2018).

[20] *Gitpython.* [Online]. Available: https://github.com/gitpython-developers/GitPython (visited on 20/09/2018).

[21] *Google code archive.* [Online]. Available: https://code.google.com/archive/ (visited on 25/10/2018).

[22] 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.

[23] 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.

[24] 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.

[25] *Homebrew core repository.* [Online]. Available: https://github.com/Homebrew/homebrew-core (visited on 31/10/2018).

[26] V. Jacques, *Pygithub*, PyGithub. [Online]. Available: https://github.com/PyGithub/PyGithub (visited on 20/09/2018).

[27] *Json*, ecma International. [Online]. Available: https://www.json.org/ (visited on 13/10/2018).

[28] 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.

[29] 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.

[30] *Linux kernel source tree.* [Online]. Available: https://github.com/torvalds/linux (visited on 20/09/2018).

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] *Nix packages repository*. [Online]. Available: https://github.com/NixOS/nixpkgs (visited on 31/10/2018).

[37] 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.

[38] *Nvd - home*. [Online]. Available: https://nvd.nist.gov/ (visited on 29/10/2018).

[39] *Odoo repository*. [Online]. Available: https://github.com/odoo/odoo (visited on 31/10/2018).

[40] 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.

[41] 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. J. 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.

[42] *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).

[43] *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).

[44] *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).

[45] 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.

[46] 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.

[47] 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).

[48] 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.

[49] *Retire.js*. [Online]. Available: https://retirejs.github.io/retire.js/.

[50] 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.

[51] *Safety - python dependency checker*, pyup. [Online]. Available: https://github.com/pyupio/safety.

[52] 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, ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2340398.

[53] 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.

[54] 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.

[55] 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`.

[56] *Sourceforge - download, develop and publish free open source software*, Slashdot Media. [Online]. Available: `https://sourceforge.net/` (visited on 25/10/2018).

[57] *Unittest - unit testing framework*, Python Software Foundation. [Online]. Available: `https://docs.python.org/3/library/unittest.html` (visited on 14/10/2018).

[58] *Welcome to python.org*, Python Software Foundation. [Online]. Available: `https://www.python.org/` (visited on 13/10/2018).

[59] 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`.

[60] 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`.