UNIVERSITÄT KOBLENZ · LANDAU

*Bachelor thesis*

# On Correlations between Vulnerabilities, Quality-, and Design-Metrics

Antoniya Ivanova

February 26, 2019

First appraiser: Prof. Dr. Jan Jürjens
Second appraiser: Sven Peldszus

Prof. Dr. Jan Jürjens
Institute for software engineering
Institute for computer science
University of Koblenz and Landau
Universitätsstraße 1
56070 Koblenz
https://rgse.uni-koblenz.de

Antoniya Ivanova
aivanova@uni-koblenz.de
Matriculation number: 213202488
Course of study: B.Sc. Computer Science
Examination regulations: PO2012

Bachelor thesis
Topic: On Correlations between Vulnerabilities, Quality-, and Design-Metrics

Submitted: February 26, 2019

Supervisor: Sven Peldszus

Prof. Dr. Jan Jürjens
Institute for software engineering
Institute for computer science
University of Koblenz and Landau
Universitätsstraße 1
56070 Koblenz

# Honorary Declaration

I hereby declare on my honor that I have developed and written the enclosed bachelor thesis completely by myself; the thoughts taken directly or indirectly from foreign sources are identified as such.

The work has not been submitted to any other examining authority and was also not yet published.

Koblenz, February 26, 2019

Signature

## Abstract

Over the past few decades society's dependence on software systems has grown significantly. These systems are utilized in nearly every matter of life today and often handle sensitive, private data. This situation has turned software security analysis into an essential and widely researched topic in the field of computer science. Researchers in this field tend to make the assumption that the quality of the software systems' code directly affects the possibility for security gaps to arise in it. Because this assumption is based on properties of the code, proving it true would mean that security assessments can be performed on software, even before a certain version of it is released. A study based on this implication has already attempted to mathematically assess the existence of such a correlation, studying it based on quality and security metric calculations. The present study builds upon that study in finding an automatic method for choosing well-fitted software projects as a sample for this correlation analysis and extends the variety of projects considered for the it. In this thesis, the automatic generation of graphical representations both for the correlations between the metrics as well as for their evolution is also introduced. With these improvements, this thesis verifies the results of the previous study with a different and broader project input. It also focuses on analyzing the correlations between the quality and security metrics to real-world vulnerability data metrics. The data is extracted and evaluated from dedicated software vulnerability information sources and serves to represent the existence of proven security weaknesses in the studied software. The study discusses some of the difficulties that arise when trying to gather such information and link it to the difference in the information contained in the repositories of the studied projects. This thesis confirms the significant influence that quality metrics have on each other. It also shows that it is important to view them together as a whole and suppose that their correlation could influence the appearance of unwanted vulnerabilities as well. One of the important conclusions I can draw from this thesis is that the visualization of metric evolution graphs, helps the understanding of the values as well as their connection to each other in a more meaningful way. It allows for better grasp of their influence on each other as opposed to only studying their correlation values. This study confirms that studying metric correlations and evolution trends can help developers improve their projects and prevent them from becoming difficult to extend and maintain, increasing the potential for good quality as well as more secure software code.

Die Abhängigkeit der Gesellschaft von Softwaresystemen hat in den letzten Jahrzehnten erheblich zugenommen. Diese Systeme werden heutzutage in fast allen Lebensbereichen eingesetzt und behandeln oft sensible, private Daten. Diese Situation hat die Software-Sicherheitsanalyse zu einem wesentlichen und viel erforschten Themenbereich im Gebiet der Informatik gemacht. Forscher aus diesem Bereich neigen zu der Annahme, dass die Qualität des Codes der Softwaresysteme die Möglichkeit, dass Sicherheitslücken entstehen, direkt beeinflusst. Da diese Annahme auf Eigenschaften des Codes beruht, kann der Nachweis dieser dazu führen, dass die Sicherheitsbewertungen einer

Software durchgeführt werden könnte, noch bevor eine bestimmte Version des Codes veröffentlicht worden ist. Eine auf diesen Implikationen basierende Studie hat bereits versucht, das Vorhandensein einer solchen Korrelation mathematisch zu bewerten, indem sie auf der Grundlage von Berechnungen der Qualitäts- und Sicherheitsmetriken untersucht wurde. Die vorliegende Studie baut auf dieser Studie auf, indem sie eine automatisierte Methode für die Auswahl gut angepasster Softwareprojekte als Stichprobe für diese Korrelationsanalyse sucht und die Vielfalt der für sie berücksichtigten Projekte erweitert. In dieser Arbeit wird auch die automatische Generierung grafischer Darstellungen sowohl für die Korrelationen zwischen den Metriken als auch für deren Entwicklung eingeführt. Mit diesen Verbesserungen werden in dieser Arbeit die Ergebnisse der vorherigen Studie anhand eines anderen und umfassenderen Projektinputs überprüft. Sie konzentriert sich auch auf die Analyse der Korrelationen zwischen den Qualitäts- und Sicherheitsmetriken und den realen Vulnerabilitätsdaten. Die Daten werden aus dedizierten Informationsquellen zu Software-Schwachstellen extrahiert und ausgewertet und dienen dazu, das Vorhandensein nachgewiesener Sicherheitslücken in der untersuchten Software darzustellen. Die Studie diskutiert einige der Schwierigkeiten, die sich ergeben, wenn versucht wird, solche Informationen zu sammeln, und verknüpft sie mit den unterschiedlichen Informationen in den Repositories der untersuchten Projekte. Diese Arbeit bestätigt den signifikanten Einfluss, den Qualitätsmetriken aufeinander haben. Dies zeigt auch, dass es wichtig ist, sie als Ganzes zusammen zu betrachten und anzunehmen, dass ihre Korrelation auch das Auftreten unerwünschter Schwachstellen beeinflussen könnte. Eine der wichtigsten Schlussfolgerungen, die ich aus dieser These ziehen kann, ist, dass die Visualisierung von Entwicklungsgraphen bestimmter Metriken das Verständnis der Werte sowie deren Verbindung untereinander sinnvoll unterstützt. Es ermöglicht ein besseres Verständnis ihres Einflusses auf einander, anstatt nur deren Korrelationswerte zu untersuchen. Diese Studie bestätigt, dass das Studium von Metrik-Korrelationen und Evolutionstendenzen Entwicklern dabei helfen kann, ihre Projekte zu verbessern und zu verhindern, dass deren Erweiterung und Wartung schwierig wird, wodurch das Potenzial für gute Qualität sowie sicherembn Software-Code erhöht wird.

# Contents


**1 Introduction** **1**
1.1 Motivation . . . 2
1.2 Objectives . . . 4
1.3 Structure of the thesis . . . 7

**2 Foundations** **9**
2.1 Software vulnerabilities . . . 9
2.2 Vulnerability database . . . 9
2.3 Quality, security, vulnerability and their defining metrics . . . 10
2.3.1 Software quality and design . . . 10
2.3.2 Software quality and design metrics . . . 11
2.3.3 Software security . . . 12
2.3.4 Software security metrics . . . 12
2.3.5 Software vulnerability metrics - CVSS score . . . 13
2.4 Statistics . . . 16
2.4.1 Recall and precision . . . 16
2.4.2 Correlations . . . 17
2.4.3 Proving correlations . . . 17

**3 Related work** **20**

**4 Automating the extraction and evaluation of software vulnerability data** **22**
4.1 The structure of a vulnerability entry . . . 23
4.2 Relevant vulnerability data . . . 26
4.3 Defining the vulnerability metrics . . . 27
4.4 Retrieving the vulnerabilities of a project . . . 28
4.4.1 Searching for project vulnerabilities . . . 28
4.4.2 Evaluation of the vulnerability search . . . 31
4.5 Comparison to other tools . . . 34

**5 Correlations between the vulnerability, quality and security metrics** **37**
5.1 Differences from the previous study . . . 37
5.2 Replicating the results of the previous study . . . 38
5.2.1 Internal correlations of the quality metrics . . . 40
5.2.2 Correlations between the quality and security metrics . . . 46
5.3 Correlations with the vulnerability metrics . . . 49
5.4 Conclusions . . . 51

**6 How can the process of selecting and extracting the software projects to evaluate be automated?** **53**
6.1 Defining features of the projects . . . 53

6.2 Automating the selection process . . . . . . 54
6.3 Evaluation . . . . . . 56
6.4 Conclusion . . . . . . 58

**7 How does the correlation between the quality, security and vulnerability metrics behave across multiple software versions? 60**
7.1 Metric evolution trends of the quality and security metrics . . . . . . 60
7.1.1 Metric evolution trends . . . . . . 60
7.1.2 Metric evolution trends for the project Smack . . . . . . 67
7.2 Metric evolution trends of the vulnerability metrics . . . . . . 68
7.3 Conclusions . . . . . . 71

**8 Implementation 73**
8.1 The Elasticsearch database . . . . . . 74
8.2 Retrieving and storing the vulnerability data . . . . . . 74
8.3 Automating the project retrieval process . . . . . . 75
8.4 Retrieving projects with at least one vulnerability . . . . . . 75
8.5 Retrieving the project metric data . . . . . . 76
8.6 Generating the metric analysis output . . . . . . 76

**9 Summary 78**
9.1 Summary of research question 1 . . . . . . 78
9.2 Summary of research question 2 . . . . . . 79
9.3 Summary of research question 3 . . . . . . 80
9.4 Summary of research question 4 . . . . . . 81

**10 Future work 82**
10.1 Analyzing more projects . . . . . . 82
10.2 Studying more correlations . . . . . . 83
10.3 Predicting vulnerabilities . . . . . . 83

**11 Bibliography 84**

# Introduction

Over the past few decades, society's dependence on software systems has grown significantly. They have found application in areas such as aiding banking operations, managing trade markets, controlling automated military and state systems as well as healthcare. This large abundance of applications and the rising need for automation enables the software market to thrive. Every day, while using it, billions of users input their data into all kinds of different programs. While parts of this data could be publicly available, other parts of it include demographic, financial and health information, company data, private addresses and others, the likes of which require secure handling and are therefore not suited for being published. Software providers, as much as users, rely on the release of secure software, because any security breaches can potentially cost them time, money and their hard-earned reputation. Yet, most software systems give rise to design and implementation weaknesses. These weaknesses often lead to vulnerabilities that open the applications to potential attacks and misuse [Fre+06]. Consequently, a lot of studies in the field of software engineering have attempted to assess the occurrences of such vulnerabilities and predict the factors that play a role in their formation. The knowledge about these factors could enable developers to reinforce the critical parts of their applications before endangering the data integrity of the users. For example, Hovsepyan et al. [Hov+12] predict possible software vulnerabilities in the span of 19 different versions of an android mail client application. By treating source code as plain text and applying analysis techniques with machine learning, they achieved an average prediction accuracy of 87%. Some more examples are presented by [RZ13] and [Shi+11]. These and others are examined in more detail in chapter 3.

In the context of vulnerability prediction, many people tend to assume a correlation between the quality of a software system's code and the security gaps that it has. For instance, this means that they would expect a software project with "low quality" code to have more vulnerabilities than a project with "high quality" code. Some have accounted the correlation to rushed product releases, which lack the time needed to apply the best secure coding practices [Wav15]. Others have accounted it to developers who copy code from the internet without checking it for security flaws [Hat] or just to insufficient secure coding practices training [Cow15]. In this thesis, the term "code quality" refers to the internal properties of software code. That is because it can be assumed that evaluating the internal quality can be used for assessing the external quality attributes [Wie17]. For example, internal quality evaluates the code's internal object structures like the amount of lines of code per class or the relationships between classes, components and methods. The external quality attributes are as defined by ISO 9126 – functionality, reliability, usability, efficiency, extensibility and portability. A closer definition of "code quality" can be found in chapter 2. From this point forward the term quality will refer to the "internal" or object-oriented design quality of code.

It is important to note that the assumption of a correlation between the quality of the software code and the security gaps it contains is based solely on properties of the code. If it can be proven, that would mean that security assessments can be performed on unreleased and not user-tested software. The phase before the quality assurance is an especially important phase in the development cycle, because any weaknesses that are found at that point usually cost much less resources to fix. This can be seen on figure Figure 1.1 which illustrates the rise in resources (time and money) spent to fix software defects in each of the software development lifecycle phases. Taking precautions to improve the software quality in this phase would allow companies to save resources in the later phases and avoid potential future vulnerabilities.

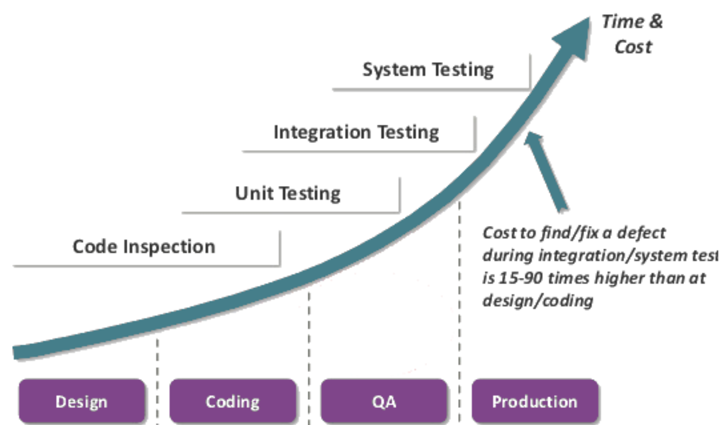


Figure 1.1: Costs of fixing a security gap in the software development lifecycle [Von16]

## 1.1 Motivation

The overall goal of this thesis is to determine whether a conclusive way to evaluate a program's security level is possible, even before it goes into the testing phase. The idea is to analyze the program code and look for any clues, which could determine the possibility for vulnerabilities to arise. The information derived from these indicators has then to be tested against some known security vulnerability data, in order to determine its practical accuracy.

This study is based on a previous study, conducted by Brigitte Wiebe [Wie17], which evaluated the relationship between the quality of object-oriented software code and its security. The previous study represents the foundation of this study and offers the fundamental term definitions and tools used.

The observations in the previous study were also based on the program code. Brigitte Wiebe used the term "security" to regard such a state of an application, which is free of risks and hazards. The term "quality" was regarded as in the previous section. Her

study delivered a tool, which statistically assesses the relationship between software quality and security. From here on, the tool is referred to as the “metric correlation analysis tool”. This tool measures what “level” of security and quality is present in a software project using a set of established metrics. These metrics assess the code in different ways and deliver some meaningful numerical values for interpretation. The tool compares the metric measurements using statistical methods and offers information about whether the quality and security aspects have a strong or weak dependency.

In her thesis, the analysis tool works with a static input list of 50 android-based projects. Analyzing further projects would increase the statistical power of the performed tests. According to [Coh92], the power of a statistical test is the probability of obtaining a statistically significant, or more reliable, result. In order to extend the tool in favour of more statistical power, the present study seeks to increase the amount of projects, that the tool accepts as input. Due to their increased amount, they should be automatically selected for importing into the tool. An important factor to consider here, is that the previous study doesn’t confirm or deny a correlation between the quality and security properties of software code, so it is possible that a bigger or different sample could allow it to do so. This remains beneficial in terms of the correlations with the metrics derived from vulnerability data, explained in the next paragraph.

The analysis tool investigates the correlation between the quality and security of software projects, based on their code. It is interesting to find out how these two factors relate to real-world data about the security state of the projects. One way to represent this state is in terms of project vulnerability statistics. They essentially give us an insight into a different kind of security aspect – the practical security. There already exists a large number of collections of information about security vulnerabilities, focusing on various subjects, that are also open for public use. Parts of this information are made available as traditional databases, mailing lists, newsletters or via newsgroups. [Sch+00] One main focus of this study is to find the right resources for vulnerability data and extract meaningful information from them. In regards to automatizing the choice of input projects for the analysis tool, this means that the choice criteria should consider the information available in the vulnerability resources. An example criteria could be whether a project has any vulnerability data posted about it at all. The real-world vulnerability information will be used to analyze to what extent the quality and security metrics of the software mirror the actual security status of the projects. For example, it could show whether a sample project, which scores lower values in its metrics for quality, would have a high amount of entries in a vulnerability database.

Finally, the possibility of trends, with respect to the correlations, could be trailed across several software versions. For example, can a project, which starts off with initially low quality and security indicators develop increasingly more bugs after each released version?

## 1.2 Objectives

This study builds upon the work of a previous study by Brigitte Wiebe [Wie17]. That study is about the correlation between the quality and security properties of object-oriented software. The previous study has produced an analysis tool, which is used to measure this correlation on the basis of code metrics. It makes use of metrics, which are already widely established and get calculated with the help of three other tools. The metrics are presented in chapter 2. As input, the tool uses a static list of 50 android applications. It gathers the results of seven quality metrics and three security metrics and performs a statistical assessment on them. It studies their correlation using two different statistical methods and delivers graphical representations of its findings. These findings show that there is a significant internal correlation between different quality metrics, but on the other hand they cannot confirm a significant correlation between quality and security metrics. These findings are evidently meaningful to the field of software engineering and a confirmation or denial of an existing correlation could be a valuable contribution to the development cycle of software projects. To address the ideas for further work, motivated in section 1.1, the present study deals with the following four research questions (RQs):

- RQ1: How can the extraction and evaluation of real-world vulnerability data about software projects be automated?
- RQ2: Can the findings of the previous study be confirmed and how do the vulnerability data metrics correlate with the quality and security metrics of the software project?
- RQ3: How can the process of selecting and extracting the software projects to evaluate be automated?
- RQ4: How does the correlation between the quality, security and vulnerability metrics behave across multiple software versions?

These questions represent the main contents of this bachelor thesis. The present section serves to clarify them in closer detail. I make use of Figure 1.2 to visualize the concept and the research questions. The reader should refer to it while reading the research question summaries to gain a better overall understanding of the goals for this thesis. RQ1 is on the bottom left of the image and it is concerned with vulnerability data extraction and evaluation. This vulnerability data is derived from software vulnerability databases. Above the databases are the software projects, which have vulnerability entries in them. These projects are automatically selected for this study in RQ3. The projects and the databases contain the concrete software versions, that the vulnerabilities are found in. On the right, the software projects flow into the metric tools of the previous study, which calculate the quality and the security metrics. For RQ2, on the bottom, the correlation between the vulnerability data and the quality and security

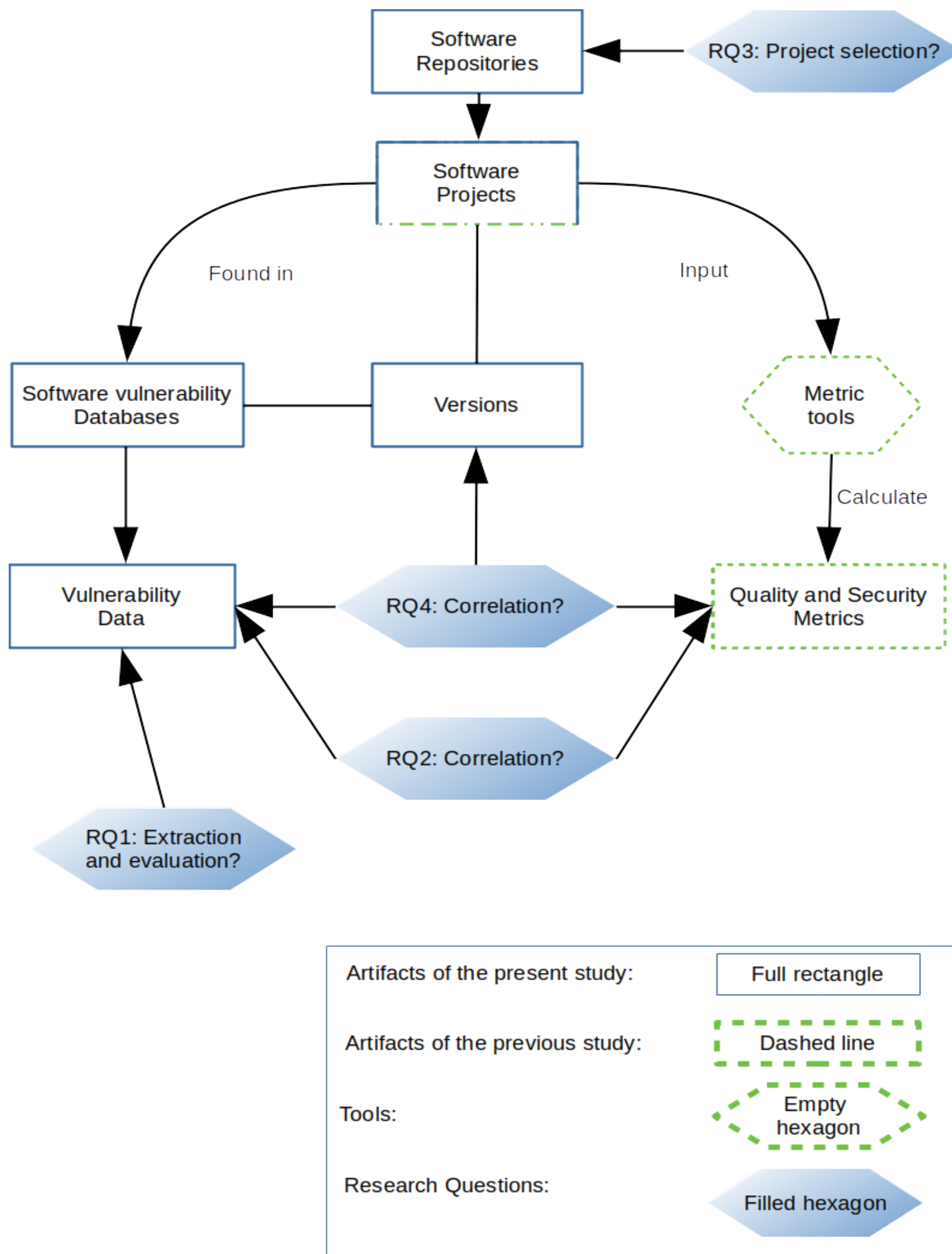


Figure 1.2: Concept of the bachelor thesis

metrics of a given software version is studied. For RQ4 the same correlation is studied across several software versions.

The shapes with green borders on the graphic represent the parts of the previous work that the present study builds upon and what is already implemented into the analysis tool. The shapes with blue borders are artifacts of the present study and the solid blue shapes represent the research questions and how they relate to the artifacts. One of the shapes is both green and blue, because it is an artifact of both studies. To explain the ideas from the graphic, we will start by going over the research questions in detail.

### RQ1 : How can the extraction and evaluation of real-world vulnerability data about software projects be automated?

The metric correlation analysis tool, which this study extends, deals with the quality and security metrics of code and their correlations. By comparing its findings to real-world vulnerability data metrics, the present study will examine further correlations. Such correlations could by way of example answer the question whether a project with bad quality or security metrics would have many real security gaps.

There are plenty resources of vulnerability data available online. They all offer different types of information. The first research question starts with finding out where to gather the vulnerability data for the future analysis from. The source has to provide up-to-date and reliable information about the software vulnerabilities of many projects.

After finding a fitting source, the focus shifts on extracting appropriate information from it. This information will be used to calculate meaningful metrics that describe the real-world security status of a given project in a certain version. These metrics will be needed to assess a correlation between them and the quality as well as the security metrics, analyzed in the previous study. Altogether, this question involves retrieving the vulnerability data and working out in addition to applying appropriate metrics to it for the correlation analysis of RQ2 and RQ4. For example, the amount of bugs submitted per project version could be a relevant metric.

### RQ2: Can the findings of the previous study be confirmed and how do the vulnerability data metrics correlate with the quality and security metrics of the software project?

RQ2 is a central deliverable of this study. Its main goal is to attempt to reiterate and confirm the findings of the previous study and is to analyze the correlation between the vulnerability, code quality and security metrics of the software projects. It is also very important to see if this study can verify the findings of the previous study with a different sample set. The metric correlation analysis tool can already calculate the

quality and security metrics (as shown in Figure 1.2). This means, it remains to be extended to evaluate the vulnerability metrics. The tool can also perform the correlation analysis between the metrics. It outputs the statistic information, as well as a graphical representation of it, allowing for an easier discussion of the results. For this question, the vulnerability metrics have to be included in the correlation analysis of the tool. Their relationship with the other metrics has to be investigated and the significance of the outcomes has to be discussed.

**RQ3: How can the process of selecting and extracting the software projects to evaluate be automated?**

The previous method of project selection for the calculations was manual. This question is about defining a good way to select projects automatically, so that the tool can utilize them, and implementing this functionality into the tool. Defining the right projects to use is key to attaining meaningful results from the calculations in the study. For example, this means that they need to use the build-tool Gradle, because it is required by the tool for the metric calculations. There are also other qualities, that should be considered in the selection process. A conceivable one is, for example, the selection of projects with present entries in the vulnerability resources. The reasoning behind this is that most projects probably have at least some vulnerabilities. Intuitively, there is a better chance that no one submitted them into the database, compared to them being 100% gap-free. The automated approach for mining the software repositories will make this method more widely applicable. Solving this question has to involve both the quality and security metrics as well as regarding the vulnerability data metrics aspect. It requires a thorough definition of the selection process.

**RQ4: How does the correlation between the quality, security and vulnerability metrics behave across multiple software versions?**

The fourth question regards the correlation between software quality and security together with the vulnerability metrics, as discussed in RQ2, but seeks to determine how they behave across several versions of the same software. A way to gather pairs of projects with vulnerabilities for every version has to be found and their correlations should be analyzed as in RQ2. The evolution of the correlations is to be discussed. Again, the findings of the previous study should be verified.

## 1.3 Structure of the thesis

In chapter 3, I present and summarize some of the related work in this research area. After that, chapter 2 contains the fundamental term definitions and clarifications needed in order to lay a common ground for the following chapters. From chapter 4 to chapter 7 I discuss over the solutions of the research questions in detail. These are followed by a short summary in chapter 9 of all the study's findings.

After that I go into some implementation details in chapter 8. Finally I list some of the prospects for future work, based upon this study in chapter 10.

# Foundations

This chapter sets the foundations that are needed for the understanding of this thesis. It defines all relevant terminology from the previous and present study. Understanding terms like security, quality and vulnerabilities is central to comprehending the presented thesis. Some of the terms here have varying definitions in different books so this chapter seeks to create a uniform understanding of them.

## 2.1 Software vulnerabilities

I have chosen to define the term "vulnerability" using the simple definition given by the Common Vulnerabilities and Exposures (CVE) list [CVE]. This is because the CVE list is an official list for known cybersecurity vulnerabilities and all vulnerabilities handled in this thesis are present in it. The organization behind it exists since 1999 and has managed to establish an industry standard for uniquely identifying vulnerabilities. According to them, their method is industry-endorsed via the CVE Numbering Authorities, CVE Board, and numerous products and services. They define vulnerabilities as "weaknesses in the computational logic (e.g, code) found in software and hardware components that, when exploited, result in a negative impact to confidentiality, integrity, or availability."

Essentially, software vulnerabilities are any kinds of weaknesses, presented by software code, which can potentially be exploited by an attacker to perform unauthorized or harmful actions.

## 2.2 Vulnerability database

A vulnerability database contains entries of vulnerabilities for software projects. These vulnerabilities are given a unique identification string. The vulnerabilities are analyzed by security professionals and filled out with some descriptive information like the product and vendor names for the project that they appear on. This information includes the vendor of the software product, the product name, the version that the vulnerability is found on as well as the release date of the version. A special score, which speaks about the severity of the vulnerability, the environment it requires in addition to the actions from the attacker and user that might be required, are included as well. The mentioned score is called the common vulnerabilities scoring system (score) or CVSS and is considered as an essential vulnerability metric. It is described in detail in subsection 2.3.5. The present study uses vulnerability databases to get information about the existing vulnerabilities of the studied projects and their CVSS score.

## 2.3 Quality, security, vulnerability and their defining metrics

To evaluate the security and quality properties of software code the previous study suggests the use of metrics. A metric is a mapping of a specific property of a software system to a concrete numerical value. Software systems can be evaluated by different criteria with the use of metrics and thus become comparable with other systems.

### 2.3.1 Software quality and design

The previous study uses the definition of software quality as in "internal quality" [Kan02; BBL76]. Internal quality is the quality of the internal object structures in object-oriented programming. This includes the relationships between classes, components and methods. Some key terms related to it are cohesion – how many different actions can a single class do – and coupling – how dependent are two separate classes on each other. Ideally, good software design should have high cohesion and low coupling. Internal quality is based solely on the properties of the software code, and not on any design models and specifications. It deals with influencing design decisions such as modularity and complexity, and gives the opportunity to measure as the existence of anti-patterns. These aspects are defined as follows:

- Modularity – Modularity is a quality characteristic of software code. A modular program is a program constructed of pieces (modules), which are self-contained and organized in a stable structure. A modular software construction produces a system made of autonomous elements connected by a coherent, simple structure. [Ame89]
- Complexity – Together with modularity, complexity is a crucial property of good-quality software code. Complexity describes the interactions between a number of entities. They have to be simple and kept as low as possible, to ensure that with a growing number of entities, the system doesn't become hard to maintain. Modularity and the definition of restricted communication mechanisms reduce complexity. Lower complexity makes software more maintainable. [Ame89]
- Anti-patterns – Anti-patterns are widespread problem solutions, which have established negative consequences [Wil98]. There are two things that distinguish an anti-pattern from a bad habit or bad practice:
  1. A common action, process or structure that appears to be effective, but turns out to have more negative consequences than it has positive ones.
  2. Another existing solution that is well-documented, repeatable and proven to be effective.

  The anti-pattern knowledge resources for this study are [Pel+16; Bro+98].

### 2.3.2 Software quality and design metrics

The following is a list of the software quality and design metrics, used in this study. A fundamental resource of further information of all these metrics is [CK94].

- Lines of code per class (LOCpC) – The amount of lines per class in the project. A higher value could point to worse quality, because a class can become unreadable and incomprehensible after a certain length is reached. Such code is hard to extend as well as maintain.

- Logical Lines Of Code (LLOC) – LLOC is language specific and measures the number of executable statements in the code, as defined by the specific language. This metric is similar to the simple LOCpC, except it ignores irrelevant formatting and style conventions, focusing on more relevant statements instead. It is also targeted at the program as a whole.

- Weighted methods per class (WMC) – The sum of complexities of all class methods. This value indicates how much effort is required to develop and maintain a particular class [MSA08]. In the previous thesis, the cyclomatic complexity sum was used to compute this. It uses the McCabe cyclomatic complexity number. This approach is a popular complexity function for calculating WMC, because it combines object-oriented class structures with the traditional sense of complexity [Mcc76]. Many small methods as well as a few complex methods would result in a bad WMC value. Classes with high WMC value are very application-specific and are likely non-reusable. If a class has a higher WMC value, other classes will also have more dependencies to it. If these classes get inherited, a high coupling takes place. This leads to a bad object-oriented design and a hard maintenance, which means worse quality.

- Coupling between objects (CBO) – The number of classes on which the current class depends [Mcc76]. Classes can depend on each other in many ways, for example through inheritance. If two classes belong to the same module, they are not calculated in the CBO value. Considering a single module, it is better to have high cohesion, as this speaks for better quality. Across modules, however, the classes have to be as independent as possible. A higher CBO value goes against the principle of low cohesion. This means the classes are less reusable and the code modularity is reduced. Because many dependencies between classes lead to probable mistakes when updating or adding new code, a high CBO value indicates complex code. Such code is hard to test as well as extend and indicates bad quality.

- Lack of cohesion in methods (LCOM) – The number of pairs of methods in a class that don't have at least one field in common minus the number of pairs of methods in the class that do share at least one field [CK94]. Classes with a high LCOM value consist of incohesive methods. Methods in a class, whose functions have nothing in common and are completely independent, make the class code more

complex. They require discrete understanding. Good software designs need to have a high cohesion and clearly show the dependence between the methods of the class. A high LCOM value goes against this principle and signifies worse quality.

- Depth of inheritance tree (DIT) – The maximum length of an inheritance path of a class [Mcc76]. A long inheritance path goes against the object-oriented principle of "composition before inheritance". A high number of inherited properties makes a class more complex. Classes with a high DIT are hard to assess in isolation and have a higher maintenance cost. This speaks for bad quality.

- Lines of duplicate Code (LDC) – Amount of identical code lines that are found in different expressions [Fow02]. Duplicate code enlarges the classes unnecessarily and makes maintenance work harder. Every duplicate has to be changed separately, which could also lead to bugs and make the process even harder. By minimizing duplicate code, code quality can be supported. A high LDC value goes against this concept.

- Occurrences of the Blob Anti-pattern (BLOB) – The blob anti-pattern occurs whenever there is a single class that controls a big part of the application process and communicates to a lot of data classes [Pel+16; Bro+98]. This class is known as a so-called "god class" [Wil98]. The BLOB anti-pattern shows a bad separation between process and data, thus it doesn't follow the principle of object-oriented design. God classes often have more than 60 attributes or methods and quickly become unreadable. God classes make code hard to reuse and slow down maintenance. Again, this goes against the principle of good code quality.

### 2.3.3 Software security

The term software security describes the idea of engineering software so that it continues to function correctly under malicious attacks. Software problems like implementation bugs and design flaws such as inconsistent error handling are often the cause of security issues. This is made possible because attackers can misuse them in order to hack into the systems and exploit them for malicious purposes. Complex and extensible software is possibly even more vulnerable to these attacks, as the security holes become more and more common. This assumption is one of the focal points of this thesis. [McG04]

### 2.3.4 Software security metrics

Similar to the object oriented design metrics, security metrics have been developed to quantify the security properties of software projects. Security has many aspects and a wide range of metrics is best suited to give a complete image of it. For this study, two descriptive security metrics are chosen. In fact, these metrics can also be described as visibility metrics, because they both have to do with managing the optimal visibility of the code components. The metrics are defined as follows:

- Inappropriate Generosity with Accessibility of Method (IGAM) – IGAM represents the ratio of methods that have a bigger access modifier than the minimal, compared to the total number of methods in an application [ZS12]. The minimal access modifier is essentially the one that allows only for such access that is crucial to the functionality of the application. This metric is meant to represent what parts of the projects are too accessible and therefore are less secure methods in their representative classes.

- Inappropriate Generosity with Accessibility of Types (IGAT) – IGAT represents the ration of the types that have a bigger access modifier than the minimal, compared to the total number of types in an application [ZS12]. Again, this represents which types are too accessible, allowing for security gaps to arise.

There is a certain limitation to the choice of these two metrics, as they describe software security from the sole aspect of visibility, when there are a few other aspects which could also be studied. The choice of other metrics and metric calculation tools for them is not a goal for this study.

### 2.3.5 Software vulnerability metrics - CVSS score

The software vulnerability metrics are classified separately from the security metrics, because there is one important difference between them. The software vulnerability metrics can not be calculated via static code analysis, but require released and vulnerable software to be analyzed instead. Apart from that one difference, the vulnerability metrics are an important descriptor of software security. They can potentially grade the software security much better than the static code analysis metrics, because they represent the real-world “is” state of the projects.

The common vulnerability scoring system score (CVSS score) is an open and standardized method for rating software vulnerabilities in a metric definition and calculation [Wan+09]. This scoring system provides a way to capture the principal characteristics of a vulnerability, and produce a numerical score reflecting its severity, as well as a textual representation of that score. It presents a framework for assessing and quantifying the impact of software vulnerabilities. This framework has three very important benefits:

- It is standardized and application-neutral, so it can be applied to any software.
- It is contextual, so it represents the actual risk a vulnerability poses.
- It is open to public use and documented.

There are two relevant versions of the score – versions 2 and 3. CVSS3 is a newer version created in 2015, which is designed to make it more accurate. In the next few paragraphs I present the CVSS2 properties. They are highly similar to the CVSS3 properties with a few differences. Those differences are explained at the end of the subsection.

CVSS scores are composed of three categories of metrics – base, temporal and environmental. The base group represents the properties of a vulnerability that doesn't change over time, such as the degree to which the vulnerability compromises the integrity of the system. The temporal group represents the properties of a vulnerability that do change over time, such as being officially patched. The environmental metrics represent the properties of a vulnerability in regards to the users' computational environments, such as potential for service loss [MSR06]. These groups and their composing metrics are represented visually in Figure 2.1.

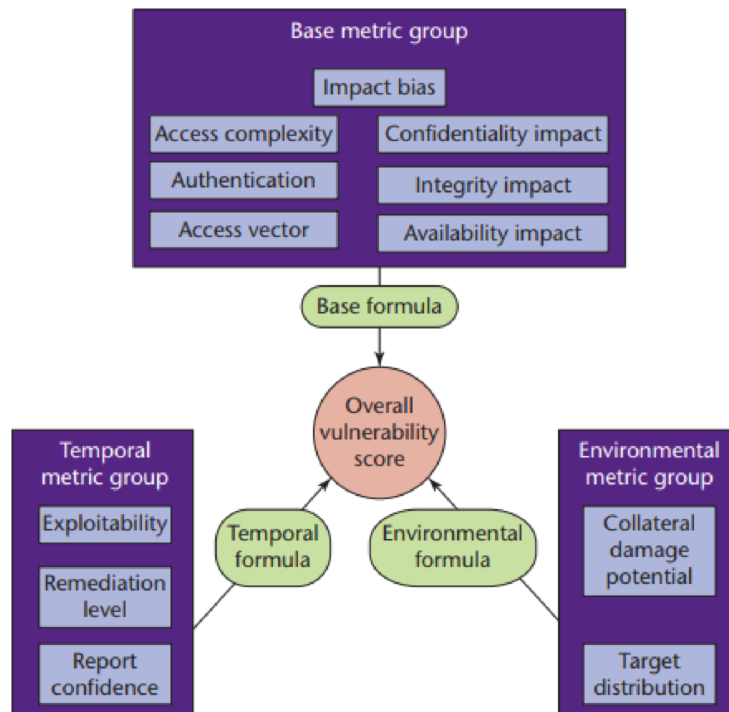


Figure 2.1: The CVSS metric groups [MSR06]

In the following listing, I present the metrics from all three metric groups and what they measure.

1. Base metrics
    - Access vector – Is the vulnerability locally or remotely exploitable?
    - Authentication – Does reaching the vulnerable spot require some form of authentication by the attacker to the operating system or application? (required/not required)
    - Access complexity – How difficult is it to exploit the vulnerability? (high or low)
    - Confidentiality impact – The potential impact of unauthorized access to the system's data. (none/partial/complete)

- Integrity impact – The potential of unauthorized modification or destruction of the system's files or data. (none/partial/complete)
- Availability impact – The potential impact of the system or data being unavailable. (none/partial/complete)
- Impact bias – Is there a property, to which the score should convey a greater weighing? (confidentiality/integrity/availability) Depending on the type of software one of these properties might be especially important in comparison to the others.

2. Temporal metrics
   - Exploitability – The current state of the vulnerability's exploitability. (unproven/proof-of-concept/functional/high)
   - Remediation level – The level of mitigating controls currently existing for the vulnerability. (official-fix/temporary-fix/workaround/unavailable)
   - Report confidence – The credibility of the vulnerability details. (unconfirmed/uncorroborated/confirmed)
3. Environmental metrics
   - Collateral damage potential – The degree of loss to information, systems or people. (none/low/medium/high)
   - Target distribution – Percentage of systems that could be affected by the vulnerability. (none/low/medium/high)

Information about the metrics is found in [MSR06]. The calculation of the final CVSS score precedes through using all three groups of metrics. It is a complex calculation that gives a thorough expression of how "severe" a vulnerability is, regarding its project and its environment. The calculation can be found on the web page of first.org [Fir].

In CVSS3, all three metric groups, the Base Score, the Temporal Score and the Environmental Score remain the same, but new metrics such as scope and user interaction are added. They are added to the base metric group.

- Scope – Whether the vulnerability is able to compromise a component other than the originally vulnerable component. (unchanged/changed)
- User interaction – Whether the user needs to do something in order to activate the vulnerability. (required/none)

The authentication metric was changed to privileges required.

- Privileges required – Whether the attacker requires any form of authentication to exploit the vulnerability. (none/low/high)

CVSS by nature does not account for chained vulnerabilities — that is, a series of vulnerabilities that when combined are capable of compromising a system. Certain CVSS scores may not always accurately reflect the severity of a particular vulnerability due to the context of the vulnerability within an organization. However, CVSS3 handles this better with the new base metric *scope*. If the value for scope is higher, not only is the Base Score increased, but the Confidentiality, Integrity and Availability impact is evaluated according to the impacted component [BRI16].

## 2.4 Statistics

This study is about assessing whether code quality and security, as measured by certain metrics, can affect the occurrence of vulnerabilities in software systems, as measured by certain metrics. To be able to determine if there is a relationship between the three aspects and more importantly how strong this relationship is, we need a mathematical method of assessment. Statistics is a branch of mathematics dealing with data collection, organization, analysis, interpretation and presentation [Y06]. This section presents a breakdown of some of the statistical tools and methods that are needed and used in this study. This should promote the better understanding of the thesis methods and result reviews.

### 2.4.1 Recall and precision

The recall and precision method is used in chapter 4 to represent how accurate and efficient the search for software vulnerabilities in the vulnerability resources is. Rigorously finding these software vulnerabilities as accurate as possible is important, because they are the foundation to constitute the vulnerability metrics that are needed further in this thesis. An accurate search for them is therefore integral to the finding of appropriate results. The keys used for the vulnerability search are derived from the project names on the GitHub platform and don't completely match the keys in the vulnerability databases. Using recall and precision allows us to see how well this search turns out given the different keys and the optimizations presented in chapter 4.

For every vulnerability search, there are four possible distinctions: a result was retrieved or it wasn't as well as it being relevant or not relevant. For a given retrieved set:

- Recall – the number of relevant items that were retrieved as a proportion of all relevant items. Recall represents the effectiveness of the retrieval. It is a measure of retrieving the most relevant items in a result set. One-hundred percent recall could be achieved if the whole vulnerability database is searched for the vulnerabilities of a given product (name and vendor retrieved from GitHub), and all relevant results are discovered.
- Precision – the number of relevant items that were retrieved as a proportion of all retrieved items. Precision represents the effectiveness of excluding all non-

relevant items from the retrieved set. It is a measure of the purity of the retrieval performance.

The trade-off in achieving high precision and recall is that usually:

- If a search is extended to retrieve more items, recall is increased. This allows for a higher number of non-relevant items to enter the retrieved set, therefore lowering precision.
- If a search is extended to further filter items that are relevant, precision is increased. This means that some relevant items could be skipped due to the added filtering, therefore lowering recall.

While one-hundred percent recall and precision score is ideally possible, it is rare in practical situations. In those cases it is best to keep them both as high as possible up to the point when one of them lowers the other. [BG94]

### 2.4.2 Correlations

A correlation is a measure of the linear relationship between two quantifiable variables. Quantifiable variables have a specific numerical value. In this thesis the metrics represent the quantifiable variables. A correlation is used to test whether there is any relationship between two variables. A relationship means that if the value of one of the variables changes, the other value will also changes in a very specific way. For example, does a high security metric (high = bad) imply a high vulnerability metric (high = bad)?

- Positive correlation – A positive correlation occurs whenever increasing the value of one of the variables also causes the value of another variable to increase. For example, the previous study was partly focused on studying the internal correlations between the quality metrics of projects and discovered that many of the quality metrics correlate positive to each other. The strongest correlation was between the LOCpC and WMC metric. This implies that together they are a coherent representation of a project's quality.
- Negative correlation – A negative correlation occurs whenever increasing the value of one variable causes the value of another variable to decrease. For example, the previous study discovered a slightly negative correlation between the quality metric CBO and the security metric IGAM. This is a rather unexpected result as both the present and the previous study go by the assumption that quality and security metrics have a positive correlation.

### 2.4.3 Proving correlations

To prove a correlation in a statistics, one needs to use a hypothesis test. A hypothesis test essentially means that we take an assumption, for example "the quality metrics of

projects have no correlation with their vulnerability metrics", and put it against the opposite assumption ("they do have a correlation") in a statistical test. The initial assumption is called the null hypothesis or $H_0$ and the opposite assumption is called the alternative hypothesis $H_1$. The goal of the test is to deny the null hypothesis and therefore prove the alternative.

Hypotheses can never be proven with absolute confidence and therefore require a certain significance value $\alpha$ to be determined. This means that there is an $\alpha$ possibility that the null hypothesis is rejected, even though its true. A smaller $\alpha$ would ensure more confident results for the test. In most cases, a value of 5% is chosen for alpha. To decide the significance of the results, a P-value is also calculated. The P-value represents the likelihood of having the specific end results, in case the null hypothesis is true. This means, that as the P-value becomes smaller, the results of the test speak more strongly against the null hypothesis. If the P-value is less than the significance $\alpha$, then the null hypothesis can confidently be rejected and the alternative hypothesis is accepted as true.

When the data, in our case the metric values for all the properties of code tested in this thesis, is gathered, it is evaluated with the assumption that the null hypothesis is true. Of course, gathering more data, means a more statistically significant result, depending on the study. Therefore this study is focused on automatizing this specific data gathering process, so more and more projects can be tested for correlations in the future. The specific procedure for the correlation analysis depends on the chosen hypothesis test. Often the so called *t-Test* is used for this matter. This test checks whether the mean values of two independent samples differ.

In this study, the correlation between the quality and vulnerability, as well as security and vulnerability metrics of projects is tested. For their analysis, a correlation coefficient has to be calculated. The coefficient describes the strength of the linear dependency between any two quantifiable variables. It's values can vary between -1 and 1. This study calculates this coefficient according to *Spearman's* ranked coefficient. The only requirement to be able to perform this calculation is that the input values can be scaled ordinal (in ranks). The calculation for this coefficient doesn't use the actual values, but these ranks. An advantage of the Spearman variant is that the outlier values can be combated, in a way that the distance between the actual values no longer plays a role. The only thing that matters is their respective order.

If the correlation coefficient r is equal to 0, this means that there is no correlation between the variables and they are completely independent. If it is larger than zero, there is a positive correlation between them. And if it is lower than zero, there is a negative correlation. The strength of the correlation can be different:

- An absolute coefficient value smaller than 0.2 implies little to no correlation
- An absolute coefficient value between 0.2 and 0.5 implies a weak correlation

- An absolute coefficient value between 0.5 and 0.8 implies a significant correlation
- An absolute coefficient value between 0.8 and 1.0 implied a high or perfect correlation

In calculating the correlation, the correlation coefficient, as well as the P-value mentioned above, are of importance. What defines as a strong or weak correlation also depends on the context and usually further calculations are needed to assess the strength of the correlation. The P-value determines the statistical significance of the correlation coefficients. It is determined by the observed correlation and the sample size which serves as a threshold for all correlation values above the absolute P-value. If the P-value is less than or equal to $\alpha$, the calculated correlation can be considered statistically significant.

# Related work

There is a lot of research published in the way of predicting the occurrence of software security vulnerabilities. Different factors are assessed in order to determine which ones have a significant influence on security gaps and how they behave in correlation to each other. Similar to the present study, Rahimi and Zargham [RZ13] propose that certain metrics of source code can be used. They apply one metric to measure code complexity and the compliance to 96 different secure coding practices in their method "vulnerability scrying". In this way they analyze four real-world applications and estimate a prediction about their security properties. They also compare the prediction scheme with real vulnerability data for the applications from a vulnerability database, called the National Vulnerability Database (NVD). They conclude, that their method can predict vulnerability trends accurately. However, they name the use of only four projects a limitation of their study. This study will therefore focus on more projects and different metrics to further analyze the correlations.

Other researchers try to predict the occurrence of vulnerabilities using vulnerability discovery models (VDMs). According to Ozment [OE07] these are probabilistic models, which operate on historical vulnerability data like a system's usage information or the date on which a vulnerability is discovered and estimate certain characteristics of the security breach discovery process. For example, such data can be found in vulnerability databases. While Rahimi and Zargham agree that VDMs can be used in predicting the discoveries, both they and Ozment discuss the many shortcomings of this technique and that it is often inaccurate. According to [RZ13] code analysis is a more reliable approach.

In [ZNW10] a team conducts an empirical study of Windows Vista and evaluates the accuracy of code quality metrics similar and overlapping with the ones used in the study preceding this one. They also use their evaluation to assess the metrics' relation to the actual vulnerabilities in the NVD. According to them it is possible to predict security gaps in this way. They advice a larger-scale study, due to their sole focus on Windows Vista and the usage of different metrics that deal with the unique characteristics of gaps and cyber attacks.

Mining vulnerability databases is mentioned in [WSS14], where researchers used prediction models based on text mining and on software metrics to find which models have a higher accuracy in evaluating the security of projects. Their findings were compared, for example, to the projects' respective entries in the NVD.

Su Zhang et al. [SO] use data mining from the NVD to predict the time until the next vulnerability of a given software application is found. They combine NVD data with different machine learning techniques to assess whether it can make accurate predictions. They found out their method had a poor prediction capability with a few exceptions.

They suggest several reasons as to why that happened. The main reason they name is the quality of the data from the NVD. They write that it is limited in the way of:

- Not having version information for each entry
- Having errors in the vulnerability information
- The vulnerability release date being dependent on the vendors' practices – they make an example of Microsoft, because they usually release their vulnerability information once per month

The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to conceive, develop, acquire, operate and maintain applications that can be trusted. They work to improve application security and offer an array of free tools and resources. They created a tool called the OWASP Dependency Check that aims at a similar target as the vulnerability gathering tool from this thesis [Pro]. Using the same vulnerability resources as in this study, the OWASP tool identifies project dependencies and checks if they have any known vulnerabilities. Their tool works by collecting information about the dependencies from the project manifest, pom-file and the package names retrieved from scanning the project's JAR files. It supports Java as well as .NET projects. The OWASP premise is that one of the reasons for less secure software is the inclusion of third party libraries that contain vulnerable code as project dependencies. Developers can analyze their usage of known vulnerable components using this tool. This thesis has a slightly different goal, but makes use of the same vulnerability resource as the OWASP tool. The thesis' tool searches for the vulnerabilities of the projects themselves and includes optimizations that are specific to its goals.

There are no current studies that use an automated approach of gathering the projects for the evaluation. The samples of most studies are very small and limited to bigger projects like the Linux Kernel and the web browser Firefox [Shi+11]. Furthermore, the other studies don't use the combination of quality and security metrics, that the tool used for this study focuses on. However, there are promising findings in the correlation of the quality and security metrics to the security database metrics, whose value is stated in many of the other cited papers.

# Automating the extraction and evaluation of software vulnerability data

In this chapter, I discuss the first research question of this thesis. The question is *"How can the extraction and evaluation of real-world vulnerability data about software projects be automated?"*. This question focuses on retrieving data about the vulnerabilities of software projects and finding a way to automatically evaluate it using the metric correlation analysis tool.

When people discover vulnerabilities in software projects, they often submit information about them to different websites or other online media. As a result there is a lot of scattered information about the security of the projects around the internet. Some organizations were created in order to unite this information. For one, the U.S. government created its own vulnerability database. In it, vulnerabilities discovered after 2002 are uniquely identified, studied and stored. This database, the National Vulnerability Database (NVD), is used for its standardized information and large size and is utilized in several studies like [SO], [Abe+06], [KRK17] and [Gha+13], which are described in chapter 3. The NVD enables the automation of vulnerability management and contains vulnerabilities from the common vulnerabilities and exposures (CVE) list. The CVE list is a list of entries for publicly known cyber security vulnerabilities. The items of the CVE list get analyzed by the NVD staff, accepted or rejected and afterwards recorded into the NVD with some representative information. This information is useful for measuring the security properties of software projects. Altogether, as of January 2019, the database contains over 108,000 vulnerabilities that have been professionally analyzed and assigned CVSS scores (subsection 2.3.5). Unfortunately, even with that amount of vulnerabilities, many vulnerabilities that could be useful for this study haven't been discovered, disclosed or otherwise reported and can therefore not be found in the NVD. Furthermore, many vulnerabilities entered in the NVD remain unconfirmed, or lack vital information. For example, some vulnerabilities have missing data about the software version they appear on. The only data that all vulnerabilities have for sure is a unique identification number from the CVE list. However, due to its popularity and size, the NVD is the chosen vulnerability resource for this study.

In the following paragraph, I start by introducing the structure of an entry in the NVD. After that I discuss what data in the NVD entries I consider relevant for this study. The next section is about the metrics that are going to be calculated using this data. After I present all the vulnerability data structure and metrics, I talk about how the vulnerabilities of the projects are actually being retrieved. There are some optimizations to the normal search process that are described, which are followed by a conclusive summary of the approach.

## 4.1 The structure of a vulnerability entry

The NVD offers an easy way to download vulnerability data as a JSON or XML file. JSON or the JavaScript Object Notation is a lightweight data-interchange format. It is easy for humans to read and write and also easy for machines to parse and generate. It has an ECMA (European Computer Manufacturers Association) established standard and is also widely used in practice as a data-interchange language [JSO]. For this study, I use the JSON files offered on the NVD website [1] for the years 2002 - 2018. These JSON files contain CVE list entries with a standardized format and information. Figure 4.1 and Figure 4.2 represent an example of what such an entry contains. Only the relevant contents are displayed, to simplify the example. All knowledge about the structure and meaning of this data can be found on the NVD and CVE websites [NVD; CVE]. In general, each vulnerability has an identifier, a product that it belongs to, vendor information about this product, a description, a CVSS score, and a few other details.

- The key *"data_type"* denotes that this is a CVE list entry. It has the MITRE *"data_format"*, because this corporation manages the CVE list, and the version of this format.
- The key *"ID"* provides the unique identifier of this vulnerability. No other vulnerability can ever obtain it. Its format is "CVE" + <year> + "-" + <number>. Here "year" stands for the year of entrance in the database for this vulnerability and "number" stands for the serial number of the vulnerability for the given year.
- The key *"vendor_name"* gives information about the identity of the software vendor, while the key *"product_name"* refers to the name of the project. The array *"version_data"* consists of several *"version_value"* keys which denote each version of the project that this vulnerability is known to appear in.
- The array *"reference_data"* contains information about resources that have reported and contain the given vulnerability as a reference, with a link to them.
- The array *"description_data"* contains the vulnerability description in plain text with information about the language it is described in.
- The key *"configurations"* defines the set of product configurations for an NVD applicability statement. It gives information about what the configuration of the program exactly is and where the vulnerability does exist (product version, vendor and others).

[1] https://nvd.nist.gov/vuln/data-feeds#JSON_FEED

- The key *"impact"* contains information about the impact scores of a vulnerability according to several different indicators. First comes the CVSS3 measurement. The sub-keys of the CVSS3 array contain all the metric values that went into it. The same is repeated for CVSS2.

With the structure of the vulnerability laid out, it becomes evident what kind of data should be extracted from each one.

```
{
  "cve": {
    "data_type": "CVE",
    "data_format": "MITRE",
    "data_version": 4,
    "CVE_data_meta": {
      "ID": "CVE-2018-0001"
    },
    "affects": {
      "vendor": {
        "vendor_data": [
          {
            "vendor_name": "jupiter",
            "product": {
              "product_data": [
                {
                  "product_name": "junos",
                  "version": {
                    "version_data": [
                      {
                        "version_value": "12.1x46"
                      }
                    ]
                  }
                }
              ]
            }
          }
        ]
      }
    },
    "references": {
      "reference_data": [
        {
          "url": "http://www.securityfocus.com/bid/103092",
          "name": "103092",
          "refsource": "BID"
        }
      ]
    },
    "description": {
      "description_data": [
        {
          "lang": "en",
          "value": "A remote, unauthenticanted attacker may be able to execute code.."
        }
      ]
    },
    "configurations": {
      "CVE_data_version": "4.0",
      "nodes": [
        {
          "operator": "OR",
          "cpe": [
            {
              "vulnerable": "true",
              "cpe22Uri": "cpe:/o:juniper:junos:12.1x46:d10",
              "cpe23Uri": "cpe:2.3.o:juniper:junos:12.1x46:d10:*:*:*"
            }
          ]
        }
      ]
    },
    "impact": {
      "baseMetricV3": {
        "cvssV3": {
          "version": "3.0",
```


Figure 4.1: An example vulnerability database entry

```
          "vectorString": "CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H",
          "attackVector": "NETWORK",
          "attackComplexity": "LOW",
          "privilegesRequires": "NONE",
          "userInteraction": "NONE",
          "scope": "UNCHANGED",
          "confidentialityImpact": "HIGH",
          "integrityImpact": "HIGH",
          "availabilityImpact": "HIGH",
          "baseScore": "9.8",
          "baseSeverity": "CRITICAL"
        },
        "exploitabilityScore": "3.9",
        "impactScore": "5.9"
      },
      "baseMetricV2": {
        "cvssV2": {
          "version": "2.0",
          "vectorString": "(AV:N/AC:L/Au:N/C:P/I:P/A:P",
          "attackVector": "NETWORK",
          "attackComplexity": "LOW",
          "authentication": "NONE",
          "confidentialityImpact": "PARTIAL",
          "integrityImpact": "PARTIAL",
          "availabilityImpact": "PARTIAL",
          "baseScore": "7.5"
        },
        "severity": "HIGH",
        "exploitabilityScore": "10",
        "impactScore": "6.4",
        "obtainAllPrivilege": "false",
        "obtainUserPrivilege": "false",
        "obtainOtherPrivilege": "false",
        "userInteractionRequired": "false"
      }
    }
  }
}
```


Figure 4.2: An example vulnerability database entry(continued)

## 4.2 Relevant vulnerability data

The previous section outlines the abundance of data available in a fully filled out vulnerability database entry. Not all of this data can be used to create relevant vulnerability metrics for the correlation analysis in this study. I have chosen the pieces of data which are most helpful for the further correlation analysis and list them with a short reasoning for their choice:

- **CVE ID:** The ID of a CVE entry is relevant as a unique identifier by which a vulnerability can be named. It is easy to track the vulnerability using only this property.

- **Vendor name:** The vendor name is relevant because a project name can not always uniquely identify a certain project online. For example a project called "Weather App" could be made by both vendors "The weather team" as well as "Weather online". In order to know which "Weather App" is being addressed here, we would need to know the vendor name as well. When trying to match the

vulnerabilities to the actual projects, it is very important to choose the right project by keeping the vendor name in mind.

- **Product name:** The product name is essential for finding the actual projects that the vulnerabilities belong to.
- **Version:** Vulnerabilities can appear and disappear from version to version. For the correlation analysis with quality and security metrics, it has to be made sure that all three aspects are compared, based on the same project version.
- **Description:** The description could be useful if it contains some data which was not entered in the other fields. Sometimes the NVD CVE entries lack their complete information and it can often be derived from other fields. Future work refinements of this thesis could find it useful to extract different kinds of additional information.
- **CVSS score:** As described in subsection 2.3.5, the CVSS score is an important metric, which will say a lot about the vulnerability state of the project. Both, CVSS2 as well as CVSS3 scores, are relevant in this regard.

## 4.3 Defining the vulnerability metrics

Based on the data, available in the NVD, I define a couple of different metrics to measure the security properties of software projects. These should represent the vulnerability aspect of the projects thoroughly and serve the correlation with the quality and security metrics as calculated in the previous thesis.

- The first choice of metric is the CVSS score. The CVSS score was designed over a few years to give a thorough overview of the severity of a vulnerability. It covers many aspects not only of the vulnerability itself but also of the environment it appears on, the user interactions and the attacker interactions it involves. It described the vulnerability in the most depth and is therefore an excellent metric to look at when discussing the results. For this reason, the extended tool extracts the CVSS2 and CVSS3 scores of each vulnerability. The vulnerability is assessed in terms of the software version it appears on. If there are several CVE entries for the same version of a product, the average CVSS2 and CVSS3 scores are computed.
- In some cases the average CVSS score metric can be heavily influenced by a few extreme results. To counteract this, a maximum CVSS score metric is added. It helps to identify how severe the worst vulnerability of the product is, as well as whether the extreme results influenced the average metric and how much they account for.
- The third vulnerability metric choice is the amount of vulnerabilities per version of a project. Projects with more vulnerabilities can intuitively be considered less secure than others. This metric does not consider the size of the project.

- One could argue that larger projects have more vulnerabilities than smaller ones, because they have more room for error. To asses if this is the case, I add an additional "vulnerabilities per 1000 lines of code" metric. This metric serves to normalize the number of vulnerabilities value and represent it relative to the project size.

## 4.4 Retrieving the vulnerabilities of a project

As previously stated, it is not easy to connect a vulnerability with a given vendor or product name to its actual project counterpart. Remembering the example of the "Weather App", one cannot uniquely identify if it was written by "The weather team" or "Weather online". Similarly, if "The weather team" has several project releases, their vendor name cannot be used to identify a specific product. Further problems arise when there are distinctions in the ways that the vendor or product name is entered into the different information resources. The NVD could write down "The weather team" as "The weather team Ltd." or something similar, while their official name on GitHub or their website could remain "The weather team" or "Team weather". This makes the automatic association between an entry about a project in the NVD and the actual project harder. A few steps are taken to give the tool a better chance at finding matches given these circumstances. These are described in the following paragraphs. Even with all the steps taken, to be able to correctly identify a vulnerability as belonging to a project is almost impossible to do for a machine. Almost every vulnerability in the NVD has a completely different vendor name from the officially accepted one for the projects. The only way to really be sure if a vulnerability belongs to a project, without matching vendor names, would be to train a machine learning algorithm to recognize this or to otherwise involve a third party check on the internet if the two vendor names are somehow synonymous.

### 4.4.1 Searching for project vulnerabilities

The first step when searching for project vulnerabilities is to save them in a local database, because programmatic searching straight from the NVD is not possible. This local database is the free Elasticsearch database. Matching the projects of the vulnerabilities in the NVD to actual project and vendor names on GitHub is not always a one-to-one (1:1) match. There is no standard that requires their names to be the exact same. A real example is a project called "roundcube_webmail" in the NVD, which is called "roundcubemail" on GitHub. To solve this, Elasticsearch supports a technique called "fuzzy search". Fuzzy searching is used whenever a string has to approximately be matched to another string. For example, a fuzzy search for "elephent", could match to the string "elephant". This technique allows the tool to find more matches between the vulnerability data and the GitHub data. An example of this follows later in Figure 4.4.1.

The last objective of this question is to provide the ability to identify the vulnerability data as belonging to specific projects on the GitHub platform. The vulnerability data from the NVD is extracted and evaluated with three different metrics. Intuitively, the project name that the vulnerability contains is the first thing to attempt matching with. This is when the problem of the non-standardized naming arises. Figure 4.3 shows the

differences in product and vendor names between the NVD (and therefore the tool's local database) and GitHub. From this sample set of data, almost none of the names are a perfect match to each other. Even if the product names are a perfect match, the vendor names often vary. It is impossible to be sure whether two software products with the same name are actually the same product, so the vendor names should be at least similar in such cases. For example "android" is recovered from GitHub as a project by "cSploit" as well as by "owntracks". In the case of "spring boot", "frog cms" and "iCMS" this means that no matching can take place, because the vendor names are too different. Several optimizations need to be applied, so that at least some of the projects can be matched to their vulnerability entries in the NVD. These are presented in the following paragraphs.

#### First optimization – Filling out missing information

The first optimization arises through the fact that the NVD sometimes misses to fill out the product, vendor and version values in their JSON data. Thankfully, this data can still be retrieved by parsing the configurations of the vulnerability. Referring back to section 4.1, it can be seen that the key "cpe22Uri" contains the following value "cpe:/o:juniper:junos:12.1x34:d10". In this case, the vendor name is "juniper", the product name is "junos", and the vulnerability appears on the project version "12.1x46". This means that for vulnerabilities which are missing this information, the configurations can still be looked at in order to retrieve it. The tool fills in its database like this wherever necessary.

#### Second optimization – Removing "_" and "-"

The second optimization that the tool makes is removing all occurrences of "_" and "-" in the local database names and also removing them from all search queries. This allows to match for example the name "spring-boot" to "spring_boot". The vendor names of spring boot are completely different, however, and provide no further leverage in matching the projects.

#### Third optimization – All entries to lowercase

The second measure, however, will not match "FrogCMS" to "frog_cms". The reason behind this is that the cases of the two strings still remain different. In order to match these results, the tool also converts all entries in its database to lowercase and converts all search queries to lowercase. Unfortunately, these products can also not be matched because of the vendor name.

| Project name (GitHub) | Project name (NVD) | Vendor name (GitHub) | Vendor name (NVD) |
|---|---|---|---|
| spring-boot | spring_boot | spring-projects | pivotal_software |
| elasticsearch | elasticsearch | elastic | elasticsearch |
| FrogCMS | frog_cms | philippe | frog_cms_project |
| wuzhicms | wuzhicms | wuzhicms | wuzhicms |
| LibreOffice/core | libreoffice | LibreOffice | libreoffice |
| iCMS | icms | idreamsoft | icmsdev |
| cmsmadesimple-2-0 | cms_made_simple | cmsmadesimple | cmsmadesimple |
| roundcubemail | webmail, roundcube_webmail | roundcube | roundcube |
| Signal-Desktop | signal-desktop | signalapp | signal |
| ocaml | ocaml | ocaml | ocaml, inria |
| telegram | telegram_messenger | drklo | telegram |
| openvpn | openvpn | OpenVPN | openvpn |
| vlc | vlc_media_player, vlc | videolan | videolan |
| etherpad-lite | etherpad, etherpad_lite | ether | etherpad |
| exiv2 | exiv2 | Exiv2 | exiv2, andreas_huggel |
| GetSimpleCMS | getsimple_cms | GetSimpleCMS | get-simple, cagintranetworks |
| zblogphp | z-blogphp | zblogcn | zblogcn |
| open-audit | open-audit | Opmantek | open-audit, opmantek |
| QuickApps | quickapps_cms | jonlyles | quickappscms |
| xiunobbs | xiuno_bbs | xiuno | xiuno_bbs_project |
| okhttp | okhttp | square | squareup |

Figure 4.3: Names on the NVD versus names on GitHub

### Fourth optimization – Fuzzy searching

A more intricate solution needs to be found in the case of “cmsmadesimple-2-0”. Even after being converted to “cmsmadesimple20”, this string won’t match “cmsmadesimple”. The fourth optimization measure makes use of the Elasticsearch fuzzy searching technique. This technique calculates string similarity based on the Levenshtein distance. The Levenshtein distance is a way to compare strings by allowing various edit operations such as deletion, insertion and substitution of individual symbols. It is defined as the minimum cost of transforming one string into another through a sequence of weighted edit operations. [YB07] For example, the string “cmsmadesimple20” has a distance of 2 from the string “cmsmadesimple”, because it requires 2 additional character insertions. The string “ephantel” has a distance of 4 from the string “elephant”, because it requires the first two and the last two characters to be edited.

Elasticsearch offers to search for a string with a maximum distance or fuzziness of 2. The tool implements the search queries with all distances from 0 to 2, but queries with a fuzziness of 2 retrieve the most results. For example, this allows it to discover that “cmsmadesimple20” and “cmsmadesimple” are actually the same project. However, this comes at a price. For example, vulnerabilities of a project like “openvpn” can get

matched to a project called “openvmf”. This means that a fuzziness of 2 may not always be optimal. This can be assessed by measuring whether the amount of true matches outweighs the amount of false positive matches. A solution that Elasticsearch offers, and the tool makes use of in its “auto” setting of fuzziness. The automatic fuzziness setting sets the fuzziness to 0 for strings of 1 or 2 characters, to 1 for strings of 3, 4 or 5 characters and to 2, for strings of more than five characters. This is important, because a project with the name, for example, “ab”, with a static fuzziness of 2, could be wrongly associated as a project named “cd”. Replacing more characters should really only be an option for longer strings.

### Fifth optimization – Even fuzzier searching

Another kind of problem arises when trying to match the project “roundcubemail” to the project entry in the NVD “roundcube_webmail” / “webmail”. These projects have very different project names in the NVD and on GitHub. To solve this, the tool makes use of the vendor name. These two projects have the same vendor “roundcube”. First, the tool searches for a fuzzy match of the vulnerability using its product name. After that, it searches for a fuzzy match of the vulnerability using its vendor name. Of course, the vendor “roundcube” could have several different projects, so it is important to refer back to the product name. If some vulnerabilities do get discovered for the same vendor, the Levenshtein distance is used yet again, to see if a looser match of the product names is possible (with a distance larger than 2). The product names are instead matched with a maximum distance equal to half the size of the search term. This number was chosen through a series of tests, as it delivered the best recall and precision. Unfortunately, the strings “roundcubemail” and “webmail” are still too different to be matched. Even the similar vendor names “elastic” and “elasticsearch” can not be matched, because their distance is more than half of the overall word length. This means that some of the entries will just not be found as there is no general way for a computer to say that these strings represent the same project.

### 4.4.2 Evaluation of the vulnerability search

To evaluate the vulnerability search and to show how well the optimizations aid in retrieving the vulnerabilities of the example GitHub projects, I created an “oracle”, which contains the desired search results for their vulnerabilities. For all the projects in Figure 4.3, I gathered the discovered vulnerabilities in the NVD by hand, and added them into a CSV file. The metric correlation analysis tool uses the results of this CSV file and compares them to its own results, retrieved by searching for the vulnerabilities of a project using its GitHub project and vendor name. In this way, I can show how the recall and precision of the search method are influenced by the optimizations in the previous sections. The search method’s recall and precision can be seen on Figure 4.4.

| | *Search keys* | | *Database keys* | | *Expected* | *Discovered* | | |
|---|---|---|---|---|---|---|---|---|
| **Project** | **Product** | **Vendor** | **Product** | **Vendor** | **#CVEs** | **#CVEs** | **Recall** | **Precision** |
| Spring boot | spring-boot | spring-projects | spring_boot | pivotal_software | 2 | 0 | 0 | 0 |
| Elasticsearch | elasticsearch | elastic | elasticsearch | elasticsearch | 11 | 5 | 0.45 | 1 |
| Frog CMS | FrogCMS | philippe | frog_cms | frog_cms_project | 11 | 0 | 0 | 0 |
| Wuzhi CMS | wuzhicms | wuzhicms | wuzhicms | wuzhicms | 20 | 20 | 1 | 1 |
| iCMS | iCMS | idreamsoft | icms | icmsdev | 16 | 5 | 0.31 | 1 |
| CMS Made Simple 2.0 | cmsmadesimple-2-0 | cmsmadesimple | cms_made_simple | cmsmadesimple | 86 | 86 | 1 | 1 |
| Roundcube Webmail | roundcubemail | roundcube | webmail, roundcube_webmail | roundcube | 60 | 8 | 0.13 | 1 |
| Signal desktop | Signal-Desktop | signalapp | signal-desktop | signal | 2 | 2 | 1 | 1 |
| OCaml | ocaml | ocaml | ocaml | ocaml, inria | 5 | 4 | 0.8 | 1 |
| Telegram Messenger | telegram | drklo | telegram_messenger | telegram | 2 | 0 | 0 | 0 |
| OpenVPN | openvpn | openvpn | openvpn | openvpn | 22 | 22 | 1 | 1 |
| VLC Media Player | vlc | vlc_media_player, vlc | vlc_media_player, vlc | videolan | 85 | 10 | 0.11 | 1 |
| Etherpad Lite | etherpade-lite | etherpad, etherpad_lite | etherpad, etherpad_lite | ether | 3 | 0 | 0 | 0 |
| Exiv2 | exiv2 | Exiv2 | exiv2 | exiv2, andreas_huggel | 55 | 54 | 0.98 | 1 |
| Get Simple CMS | GetSimpleCMS | GetSimpleCMS | getsimple_cms | get-simple, cagintranetworks | 20 | 17 | 0.85 | 1 |
| Z-blog PHP | zblogphp | zblogcn | z-blogphp | zblogcn | 10 | 10 | 1 | 1 |
| Open Audit | open-audit | Opmantek | open-audit | open-audit, opmantek | 9 | 3 | 0.33 | 1 |
| QuickApps | QuickApps | jonlyles | quickapps_cms | quickappscms | 2 | 0 | 0 | 0 |
| OkHTTP | okhttp | square | okhttp | squareup | 1 | 1 | 1 | 1 |
| Xiuno BBS | xiunobbs | xiuno_bbs | xiuno_bbs | xiuno_bbs_project | 2 | 1 | 0.5 | 1 |
| LibreOffice Core | core | LibreOffice | libreoffice | libreoffice | 26 | 0 | 0 | 0 |
| **Sum** | | | | | 450 | 248 | | |
| **Average** | | | | | 21,42857143 | 11,80952381 | 0,5 | 0,7142857143 |

Figure 4.4: Recall, precision and number of discovered results by the metric correlation analysis tool

A few conclusions can be drawn from the data in Figure 4.4. There are many cases in which the projects cannot be linked to each other by the tool. In the case of elasticSearch, frogCMS, iCMS, OCaml, telegram, VLC media player, etherpad lite, exiv, get simple CMS, open audit, quickApps, xiunoBBS and spring boot, this is because the vendor names are too different from each other. For some of these projects some matches are made, because they are not consistently entered in the NVD database. For example, the product exiv sometimes has the vendor exiv2, as used in GitHub as well, and other times has the vendor Andreas Huggel. It is possible that this is due to the NVD staff looking for the product names in the common platform enumeration (CPE) list, which is used to refer to IT products and platforms in a standardized way for machine processing. If they could not retrieve the name entries, it can be considered that they forged the names on their own, leading to inconsistencies in the NVD naming practices later down the line. As this can not be proven in practice it should be regarded as speculative thinking, but none the less remains a possibility.

In the case of projects like Roundcube Webmail and VLC Media Player, very few vulnerabilities are discovered, because the product names are so different, that even if the vendor name is matched they cannot be related to each other. This cannot be fixed because allowing a more loose matching of the product names would lead to a lower precision. The precision for the vulnerabilities that are matched is 100%. This comes at the price of a lower recall for some of the projects. It is clear to see that the varying vendor names make the matching task much harder than it is anticipated initially. Paired with the inconsistencies in the NVD database itself, it becomes even more unreliable. Initially this led me to believe that the matching can be done with less regard of the vendor names, but this strategy acquired too many false positives. For example, many

products on GitHub have to do with the word “android” in their product name:

- “android” by “cSploit” - the cSploit program but for Android, named just “android” in GitHub.
- “android” by “owntracks” - the owntracks Android app.
- “Xndroid” by “XndroidDev” - a proxy software for Android based on XX-Net and fqrouter.
- “Android” by “Jhuster” - example code for android applications
- “kandroid” by “keeganlee” - a learning project built on the architecture of Android.

If those projects would be matched for vulnerabilities without their vendor names, they would all appear to be the same project, but in fact they are not. This means that they would appear to have many vulnerabilities because they would have the sum of all matched vulnerabilities attached to each of them.

Another obvious problem with these names is that they are not the actual names of the products. The product names are “cSploit” and “owntracks”, but not “android”. This is an issue when mining GitHub, because one can not actually be sure what naming convention was used for the projects. Some projects will have their company name be the root folder and the project name be the sub-folder, others would take the project name as the root folder. There is no way to be sure what method was used, and it is often hard to tell this by the names of the projects. In the case of “android” this could be assumed and handled pragmatically. For the most accurate results, the GitHub vendor and product names should both be fuzzy matched to the vulnerability entries of the respective vendor and product names. It is unfortunate that only half of the expected vulnerabilities are discovered this way, but the discoveries are at least definite matches.

In the case of QuickApps, both the product as well as the vendor names for the project from both resources are similar, yet they are too far from each other in their Levenshtein distance. This distance should not be increased, as it would inevitably lower the overall precision. Finally, the case of LibreOffice could not be resolved, because GitHub delivers the name “core” for this project. The vendor names still match, yet the tool can not verify whether the project is actually the same. Clearly the fault is not always in the NVD database entries, because there are some examples of bad product and vendor naming practices on GitHub too. The real problem is that GitHub is not necessarily a place where people would carefully enter the metadata about their products. This is completely optional for the developers, but in turn makes studying the projects much harder.

To summarize, matching the vulnerability entries from the NVD to their GitHub project counterparts is a non-trivial task, but many precautions have been undertaken to

increase the precision and recall of the matching for this specific method. Unfortunately, partly due to the inconsistent naming between both resources and partly due to some lacking information and conformity on both sides, not every pair can be matched. The final search method for the tool, with the given set of projects, still achieves an average recall of 50% and an average precision of 71%. The tool will automatically be able to achieve much more if the NVD team decides to take a more consistent approach to their future entry mechanisms. The NVD database is a resource for many studies and there should be some stricter policies in place for the adding of new entries. On the other hand, the developers on the GitHub platform would also benefit from consistent naming policies, which would make their projects easier to discover.

## 4.5 Comparison to other tools

In this section I compare the most popular open-source vulnerability search tool found online to the tool created in this thesis. This comparison is meant to give the reader an idea of how the presented tool, with all the optimizations included, compares to a current state-of-the-art tool, both in its method and in its results.

The tool, that I have found for this comparison, is called "CVE Search" and is found on the GitHub platform[2]. It's creators are Wim Remes, Alexandre Dulaunoy and Pieter-Jan Moreels. This tool has been used by some representative institutions such as the Computer Incident Response Center (CIRCL) Luxembourg, which is also its sponsor. It has close to 900 stars on GitHub at the time of writing this thesis. It also has more than 1400 commits from 26 contributors. I used this tool's search API to search for the vulnerabilities of the same set of projects as in Figure 4.3 by hand. The expected results for this search were derived with the help of the same oracle as for the metric correlation analysis tool. The number of discovered results, as well as the recall and precision are shown in the following graphic.

[2]https://github.com/cve-search/cve-search

| | *Expected* | *Values for CVE Search* | | | *Values for the metric correlation analysis tool* | | |
|---|---|---|---|---|---|---|---|
| **Project** | **#CVEs** | **#CVEs** | **Recall** | **Precision** | **#CVEs** | **Recall** | **Precision** |
| Spring boot | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| Elasticsearch | 7 | 0 | 0 | 0 | 5 | 0.45 | 1 |
| Frog CMS | 11 | 0 | 0 | 0 | 0 | 0 | 0 |
| Wuzhi CMS | 14 | 4 | 0.22 | 1 | 20 | 1 | 1 |
| iCMS | 11 | 1 | 0.09 | 1 | 5 | 0.31 | 1 |
| CMS Made Simple 2.0 | 82 | 0 | 0 | 0 | 86 | 1 | 1 |
| Roundcube Webmail | 59 | 0 | 0 | 0 | 8 | 0.13 | 1 |
| Signal desktop | 2 | 0 | 0 | 0 | 2 | 1 | 1 |
| OCaml | 5 | 4 | 0.8 | 1 | 4 | 0.8 | 1 |
| Telegram Messenger | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| VLC Media Player | 85 | 83 | 0,97 | 1 | 22 | 1 | 1 |
| Etherpad Lite | 3 | 0 | 0 | 0 | 10 | 0.11 | 1 |
| OpenVPN | 22 | 25 | 1 | 0,88 | 0 | 0 | 0 |
| Exiv2 | 55 | 55 | 1 | 1 | 54 | 0.98 | 1 |
| Get Simple CMS | 13 | 0 | 0 | 0 | 17 | 0.85 | 1 |
| Z-blog PHP | 10 | 0 | 0 | 0 | 10 | 1 | 1 |
| Open Audit | 7 | 1 | 0,14 | 1 | 3 | 0.33 | 1 |
| QuickApps | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| OkHTTP | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| Xiuno BBS | 1 | 0 | 0 | 0 | 1 | 0.5 | 1 |
| LibreOffice Core | 26 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Sum** | 420 | 173 | | | 248 | | |
| **Average** | 20 | 8,238095238 | 0,1727777778 | 0,3276190476 | 11,80952381 | 0,5 | 0,7142857143 |

Figure 4.5: Recall, precision and number of discovered results by the open-source vulnerability search tool CVE Search compared to the metric correlation analysis tool

A few things stand out when reading this table. The first thing is that many of the searches retrieved zero results, where the metric correlation analysis tool retrieved at least a couple. This shows that while the improvements to the tool did not make it able to recognize every match, they gave it a much better chance of finding at least the more similar ones. The second thing is that wherever the searches did wield results, the precision is either 100% or very close to that. These two observations tell something about the way that the search is made. It is meant to be very accurate and therefore it is restrictive in terms of its search keys. Much like in the metric correlation analysis tool, that comes at the price of the recall values in those cases. Keys that strongly varied between GitHub and the NVD also delivered close to no results. The average recall for CVE Search lies at 17%, with a precision of 32%. In comparison, the metric correlation analysis tool achieves an average recall of 50% and an average precision of 71%. As explained in subsection 2.4.1, a high recall and precision are hard to balance at the same time and one usually comes at the cost of the other. The CVE Search tool and the metric correlation analysis tool both offer a high precision at the cost of a lower recall. There is just no simple way around this, given all the circumstances presented in subsection 4.4.2.

It seems that no tool is currently equipped well-enough for the purposes of this thesis. The exact matching with the GitHub project repository names is a challenge that re-

quires the input from more future work. The outcomes might be different, if the names for the search would be derived from another resource, but this is not useful for the present study. Of course, a higher amount of results would give a bigger basis for future studies, and that is an important goal to aim for in the context of vulnerability prediction.

As with the metric correlation analysis tool, CVE Search also failed to find any vulnerabilities for harder projects such as Libre Office and Roundcube Webmail. Given the significant difference in their NVD and GitHub names, it is unlikely that they can automatically be discovered by any tool without significantly lowering the overall precision for the search. For a search to find them, it would have to be more inclusive than the metric correlation analysis tool. Such a tool would find too many false positive results to be reliable for research. An example for this is given in subsection 4.4.2. There could be another way of matching such results, which does not involve any fuzzy searching techniques. Possibly, such a solution would require some kind of additional metadata to be entered, or machine learning to be utilized in some way.

# Correlations between the vulnerability, quality and security metrics

The second research question is about analyzing the correlations between the metric values that the tool gathers and discussing the results. For the purposes of this question, the tool generates correlation tables for all metric results and correlation graphs for every pair of metrics automatically. While the correlation tables were existing since the creation of the previous study, the automatically generated graphs are a new addition in this study and aid the automatizing of the results analysis. In the future, the metric correlation analysis tool can be ran on an increasing number of projects with great ease and deliver many visual representations of its findings to aid the analysts.

The previous study used the tool to assess the correlations between the quality and security metrics. A fundamental part of the present study is to reiterate and attempt to confirm the results of the previous study. This is the mandatory first step for this research question. There are several reasons why this is beneficial. First, the present study has automatically picked and analyzed a completely different set of projects. These projects are no longer smaller and less known android applications, but are indeed some of the most popular projects on GitHub that fulfill all the study prerequisites. Second, reiterating the correlation analysis from the previous study with a very high degree of automation and a different data set will do a lot in the way of confirming its findings. The condition here is that the results have to be very similar. If they are different, more future studies might be needed to be able to make confident assumptions on this topic. Additionally, the present study adds the aspect of vulnerability metrics.

All metric values that were considered for this specific research question were calculated based on the newest software version of the projects. If the newest version caused some error of calculation, the next working version was assumed. There is a trend overview across several software versions in chapter 7. The following sections are focused more on verifying the findings of the previous study. This means that they give insight about the internal correlations between the quality metrics, and the correlations between the quality and security metrics. The benefit to that is the different data set that the correlations were analyzed from, and the easy scalability of the extended tool.

## 5.1 Differences from the previous study

Before we discuss the results replication in regards to the previous study, I would like to list some of the changes that were made between both studies. The biggest difference is the type of projects that were analyzed. In the previous study, 50 android applications were used as the input for the correlation analysis. They were mostly smaller projects. In this study, we are looking at 33 of the most famous Java + Gradle projects on GitHub, which are likely to present a broader spectrum of quality indicator values.

Another difference related to the removed limitation for android projects is that the present study has removed one of the metrics from the previous study, that only made sense when applied to android projects. On Figure 5.2 the MPR metric is listed. This metric was removed when broadening the scope of this study. It was essentially related to how many permissions a user gives to an application as opposed to how many it actually needs.

Finally, apart from the new vulnerability metrics, the LLOC metric was added in this study. It turns out to be a relevant addition, judging by the amount of correlations it has with the other metrics. The following sections describe these in some more detail.

## 5.2 Replicating the results of the previous study

The tool delivers its results in the form of a correlation table using the Spearman correlation method. The correlation values range between -1 and 1 for negative as well as positive correlations. Weak correlations are ranging, as defined in section 2.4, between the absolute values of 0.2 and 0.5, medium correlations between the absolute values of 0.5 and 0.8 and strong correlations between the absolute values of 0.8 and 1. If we account the number of projects that went into the calculations and the significance value $\alpha$ , we can look up the critical (P) value for the correlations in the table. The tool was able to assess 39 projects and $\alpha$ was chosen to be 5%, which sets the P-value at 0.317 in this case. Any value bigger or equal to the absolute P-value can be considered statistically significant. The table looks as follows:

| | LOCpC | LLOC | CBO | LDC | WMC | DIT | LCOM5 | BLOB - Antipattern | IGAM | IGAT |
|---|---|---|---|---|---|---|---|---|---|---|
| LOCpC | 1 | | | | | | | | | |
| LLOC | 0.52 | 1 | | | | | | | | |
| CBO | 0.49 | 0.82 | 1 | | | | | | | |
| LDC | 0.6 | 0.41 | 0.28 | 1 | | | | | | |
| WMC | 0.91 | 0.47 | 0.51 | 0.53 | 1 | | | | | |
| DIT | 0.13 | 0.53 | 0.73 | -0.01 | 0.28 | 1 | | | | |
| LCOM5 | 0.4 | 0.66 | 0.44 | 0.34 | 0.44 | 0.39 | 1 | | | |
| BLOB - Antipattern | 0.53 | 0.94 | 0.81 | 0.4 | 0.44 | 0.45 | 0.64 | 1 | | |
| IGAM | -0.16 | -0.5 | -0.41 | -0.17 | -0.14 | -0.38 | -0.48 | -0.39 | 1 | |
| IGAT | -0.29 | -0.22 | -0.18 | -0.11 | -0.32 | 0.06 | -0.05 | -0.18 | 0.35 | 1 |
| | Quality | | | | | | | | Security | |

Strong correlation
Medium correlation
Weak correlation
Critical value = |0.317|

Figure 5.1: The resulting Spearman correlation table

At a first glance it is evident that all but two of the quality metrics have at least a weak correlation with each other. Furthermore, only four of them are under the critical value and therefore not significant. In the previous study, only one pair of quality metrics did not have at least a weak correlation with each other, and four of them were under the critical value. The critical value for the previous study with 50 projects was 0.28. The quality metric correlations from there are presented in Figure 5.2 and the security correlations with them are presented in Figure 5.3.

It is good to see that a very similar result was achieved with a completely different, and also relatively large, set of input projects. The main difference with the present study is that the projects studied this time were mostly bigger and very popular. They were no longer limited to android projects, but all kinds of different java applications and yet they still have quality metrics that behave in much the same way as they did in the previous study. This reiterates the solid picture of the software quality, delivered by this specific set of metrics. They give a very coherent indication of the project status and can all be calculated by the metric correlation analysis tool before the project actually releases and without causing too much effort. Developers can definitely make use of them if not necessarily to avoid vulnerabilities, then at least to ease the process of extending and maintaining their code. In this way they will assure that it has a higher longevity, saving time and money for the development of newer projects for their companies. Simpler and better structured products are also definitely much easier to comprehend for junior developers, which saves time in terms of bug fixes and training

them. Because of this, I think it is very likely that if there were more vulnerability metric results, the quality metrics would have some influence on them as well.

| | LOCpC | WMC | CBO | LCOM | DIT | LDC |
|---|---|---|---|---|---|---|
| LOCpC | X | | | | | |
| WMC | 0.86 | X | | | | |
| CBO | 0.48 | 0.67 | X | | | |
| LCOM | 0.41 | 0.64 | 0.37 | X | | |
| DIT | 0.23 | 0.26 | 0.64 | 0.01 | X | |
| LDC | 0.62 | 0.66 | 0.59 | 0.42 | 0.25 | X |
| BLOB | 0.29 | 0.47 | 0.73 | 0.34 | 0.42 | 0.51 |

Figure 5.2: The correlation table for the quality metrics from the previous study [Wie17]

The situation is different when it comes to the security metrics. The correlations with the security metrics in the present study are mostly non-existent and not significant. Surprisingly, the existing correlations are negative for all but one correlation. This is generally an unexpected result, but it is not one that is much different from the result of the previous study. The correlations for the present study are only slightly more significant. The lack of correlations is most likely due to the aspect of security that the security metrics represent. Rather than present a more rounded representation of the security aspect of the software code, they focus too starkly on the visibility-security aspect, leaving other areas uncovered. A bigger part of the security aspect can be covered with the metric correlation analysis tool by extending the vulnerability search accuracy with a more powerful matching algorithm. I don't expect the current security metrics to correlate with the vulnerability metrics, as higher visibility is only a way of misusing a pre-existing vulnerability. That is why I think that these specific security metrics have less to do with the software quality code than initially expected.

| | LOCpC | WMC | CBO | LCOM | DIT | LDC | BLOB |
|---|---|---|---|---|---|---|---|
| IGAM | -0.01 | -0.05 | -0.31 | 0.06 | -0.11 | 0.01 | -0.27 |
| IGAT | 0.04 | 0.03 | 0.00 | 0.12 | 0.15 | 0.13 | 0.14 |
| MPR | -0.02 | -0.18 | -0.39 | 0.07 | -0.27 | -0.20 | -0.26 |

Figure 5.3: The correlation table for the quality and security metrics from the previous study [Wie17]

### 5.2.1 Internal correlations of the quality metrics

In the following subsections I go over the metrics and their correlations in the order that they are presented in the columns of the table. I discuss some of the most dominant

correlations and also present the visual representations for some of the more interesting results.

## LOCpC

Starting from the correlation between LOCpC and LLOC, we can see that it is quite strong and positive with a value of 0.52. It is logical, that having more lines of code per class means, that there are more lines of code in general. The exception to this would probably be projects with a lot of in-code documentation and comments that don't count into the calculation of the LLOC metric.

LOCpC also has a quite strong correlation with the lines of duplicate code LDC with a value of 0.6. This is also easily explained with the fact, that having a lot of lines per code in general is likely to mean that some code is duplicated.

One of the strongest correlations is the one between the lines of code per class and the weighted methods per class metric WMC. In the previous study, these two metrics had a very strong positive correlation of 0.86. This correlation is almost a perfect positive correlation in the present study with a value of 0.91. This can be attributed to the fact that the calculation of the weighted methods per class metric takes the number of methods in consideration. Because having a higher amount of code lines likely implies having more methods, these two metrics strongly reflect each other.

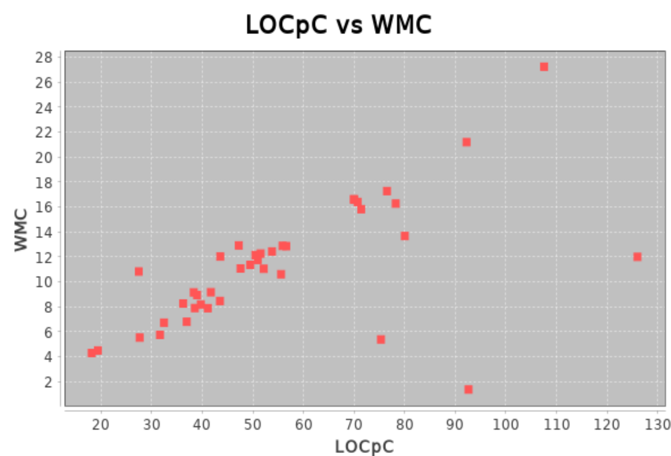


Figure 5.4: LOCpC vs WMC

The LOCpC metric also correlated quite strong with the occurrences of the BLOB-anti-pattern metric with a value of 0.53. This is because having more BLOBs in the code is likely to cause it to increase in size and influence the LOCpC metric. BLOBs are defined as very big classes that have a lot of different functionalities so this correlation is expected.

## LLOC

The logical lines of code LLOC metric has a very strong positive correlation with the coupling between objects CBO metric with a value of 0.82. I explain this phenomenon with the fact that having many lines of code in a program probably means there are many classes and that is likely to mean they have many inter-dependencies. On the graphic we can see, that as the lines of code increase, the CBO also increases steadily. This is even true for the one extreme case.

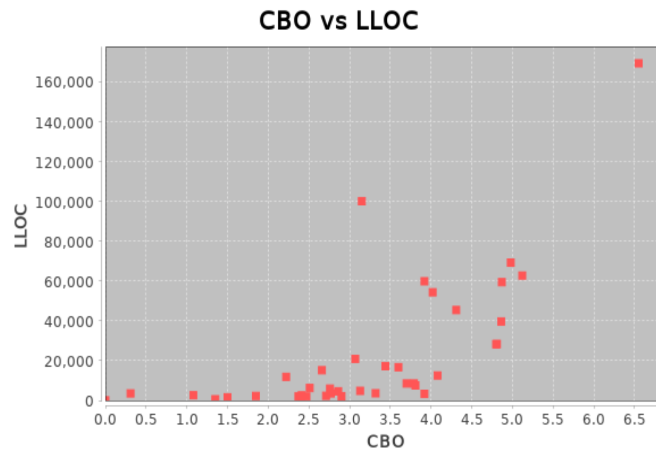


Figure 5.5: LLOC vs CBO

The logical lines of code also correlate strongly with the depth of inheritance tree DIT. Their correlation value is 0.53. This is likely because the code base is bigger and therefore there are more classes which could inherit from one-another. This goes for the lack of cohesion in methods LCOM5 as well, although the stronger positive correlation is not necessarily expected. It is likely that bigger projects are prone to more issues, like the lack of cohesion for example.

Finally, the growing number of code lines seems to be a direct cause of many BLOB anti-patterns appearing. The correlation value for these two metrics is 0.94. Programmers should be weary of this correlation as their code bases expand. For bigger projects, it might sometimes be impossible to avoid BLOBs as their functionalities grow more and more complex, so this might not always be an inherently bad thing. This would be a much worse sign in smaller, simpler projects, where there are easier ways to divide the code among more classes.

## CBO

The coupling between objects metric correlates with every other quality metric. It is related to a higher complexity, which was also discovered in the previous study. Classes with a high WMC value have many as well as complex methods and may become increasingly hard to comprehend, causing them to require more classes to be imported. It

could also be, that these classes handle many complex functions at once, which requires a lot of additional classes to be utilized. Either way both of these metrics are amplifying the bad consequences of each-other and are something developers should probably track more carefully.

The CBO and DIT metrics have a significant positive correlation in both studies too. In the present study, their correlation value is at 0.53 and in the previous study it is 0.64. I think this is because the increasing length of the inheritance tree causes the amount of classes, that the child class will depend on, to become higher.

The BLOB anti-pattern has a very strong positive correlation with the CBO. The correlation value is at 0.81. A big class will often make use of many other classes for all its different methods, increasing the coupling. This lowers the code quality after a while as it becomes harder and harder for the dependencies to be tracked and the separation of concerns principle to be applied. This could lower the quality so much that it becomes easier for newer developers to simply try to reproduce the code from scratch. The significant positive correlation can be seen on Figure 5.6 and was also found to be the second strongest in the previous study, where it had the value of 0.73.

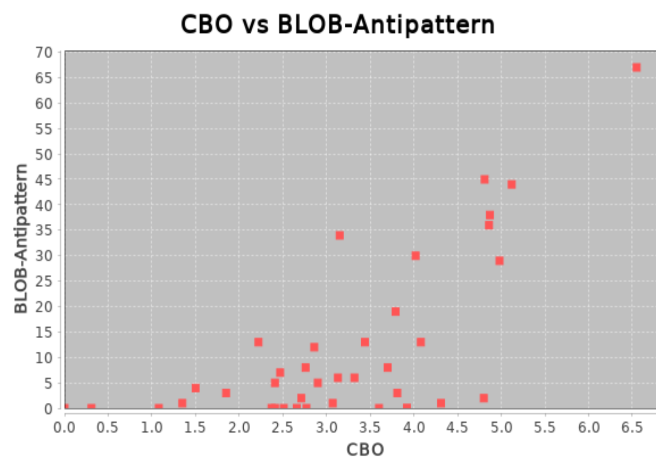


Figure 5.6: BLOB vs CBO

## LDC

The lines of duplicate code metric has a slightly stronger correlation only with the weighted methods per class WMC. Their correlation value is at 0.53. This is likely due to complicated code being reused several times. The cause might be that the code is so hard to understand, that it becomes easier for the developers who have to work on it to simply copy it, without understanding how it works or the ambition to simplify it. Simpler code could be written on the spot and it would likely look different than code with a similar purpose written in the past elsewhere in the project.

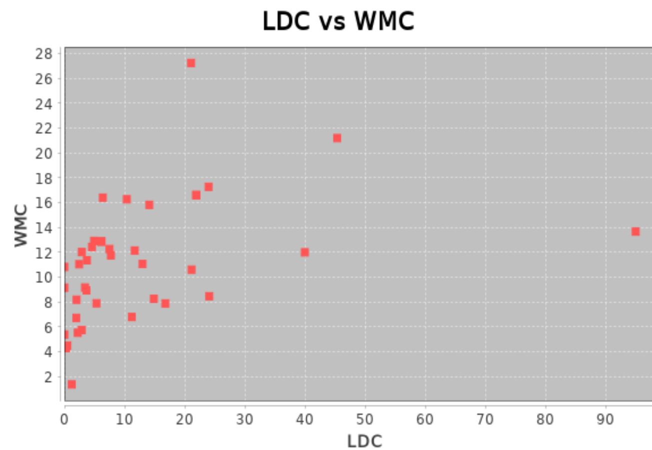


Figure 5.7: LDC vs WMC

The lines of duplicate code also correlate slightly, but significantly with the LCOM5 and BLOB metrics. The correlation values are respectively 0.34 and 0.4. With the BLOB-anti-pattern it is understandable that duplicating code could cause some classes to grow significantly in size. It is however not necessary that the code is duplicated in the same class. Developers consider the consequences that duplicating code does to the complexity and length of their classes, because these are two very important contributors to their extensibility.

## WMC

The weighted method per class metric in general correlates with every other metric. I think complexity is a big problem in software engineering, as problems get simpler to solve with experience. A lot of less experienced developers write more complex code as the concepts and tools of the trade are still new to them. This is something that they simply have to get over with experience and can hardly avoid, but it brings along many other quality issues with itself. WMC has slight correlations with DIT, LCOM5 and the BLOB-anti-pattern metrics.

WMC and LCOM5 likely correlate because having many complex methods in a single class would likely involve many variables and objects to be used. Those variables and objects would serve a very specific purpose in the complex methods and therefore should not be shared with the rest of the class methods. The length of these methods is probably why the WMC metrics also correlates with the BLOB-anti-pattern. Simpler methods are likely much shorter. The correlation value for WMC and LCOM5 is at 0.44.

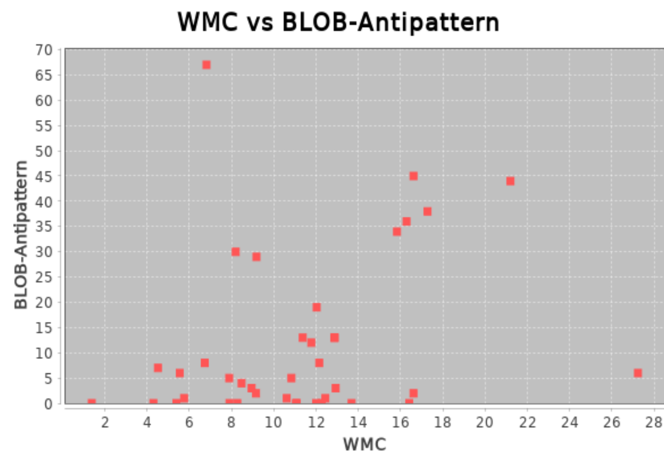


Figure 5.8: BLOB vs WMC

## DIT

The depth of inheritance tree has significant weak correlations to the lack of cohesion in methods and the blob anti-pattern. The correlation values here are respectively 0.39 and 0.45. The correlation between having deeper inheritance trees and the BLOB anti-pattern could be explained by larger BLOB classes inheriting from many other classes, as they have a lot of packed functionality. If there are many BLOB classes who all do this, the DIT would inevitably grow.

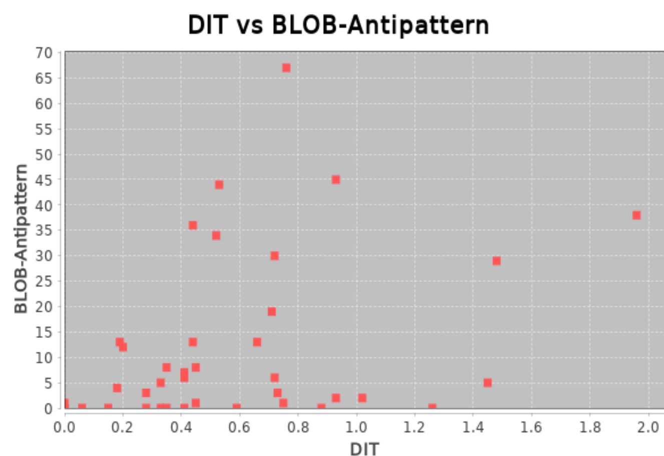


Figure 5.9: BLOB vs DIT

When it comes to the lack of cohesion in methods, there is a slight correlation with the depth of inheritance tree metric with a value of 0.39. This differs from the findings of the previous study, where these two metrics were found to be completely independent. This might be related to the fact that classes which inherit many other classes are likely to be using them for several purposes that might not have much to do with each other.

If they do not have a common purpose, these methods should not be in the same class. This increases the lack of cohesion in methods metric. I see this as an expected result but it is not a necessary occurrence, just a possible one. This might be the reason it was not found in the previous study.

### LCOM5

Finally, the lack of cohesion in methods has an expected medium significant correlation to the occurrence of the BLOB anti-pattern. Their correlation value is 0.64. In the previous study they had a lower, but still significant correlation of 0.34. The LCOM5 metric is even used in the calculation of the BLOB metric. BLOB classes are very large classes that pack a lot of functionality. While this might sometimes be necessary, it might turn out that these classes include methods that do not belong together and are meant for entirely separate purposes. It might be because the projects in the present study are larger and more mainstream that this result appears different, but I think it is definitely an expected outcome.

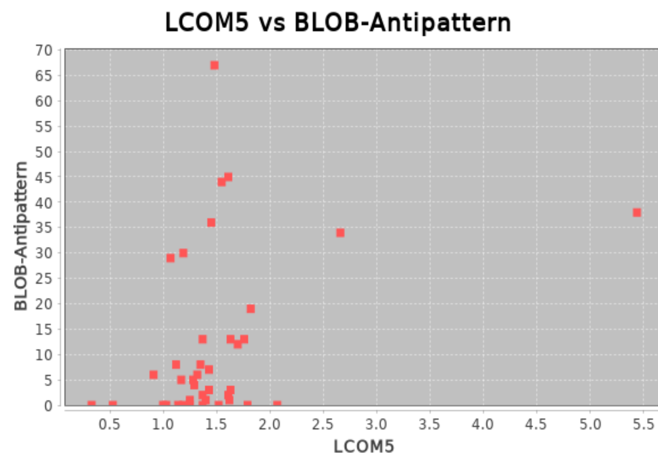


Figure 5.10: BLOB vs LCOM5

## 5.2.2 Correlations between the quality and security metrics

Altogether it seems that if anything there are only negative correlations between these two types of metrics. This is a surprising result, but only before we look at the previous study. The previous study also discovered that there are only none or negative correlations between the two aspects.

### IGAM

The inappropriate generosity with accessibility of methods metric has all negative and mostly small correlations with the quality metrics. This is unexpected, because the

initial assumption for both the present and previous study is that the quality metrics for the projects should affect the security metrics in a negative way. If we look into the correlations with more detail, however, we can see why that is not the case and notice that some correlations are actually to be expected.

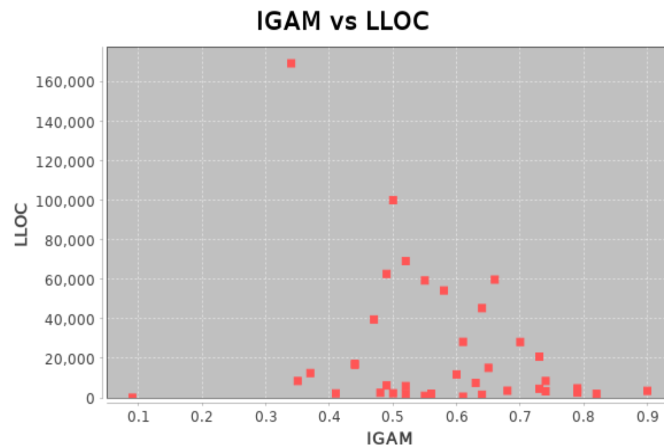


Figure 5.11: IGAM vs LLOC

On the graph for LLOC and IGAM we can see that a relatively high IGAM value can be achieved with a very low to very high LLOC score. Their correlation value is -0.5. I don't think this is an expected outcome, because one could assume that having more code in general makes a project more prone to security opening and mistakes, that will higher the IGAM score. Unfortunately, this correlation is not shown in the previous study and they cannot be compared. It is possible that developers working on larger projects are employed in bigger companies that are more careful with their software security. On the graph we can see that the smallest projects achieved the biggest IGAM values. Maybe the smallest projects are not very keen about proper secure coding practices, but more about showing some coding examples and achieving their purpose fast.

IGAM also has stronger negative correlations of -0.41 with the CBO metric and of -0.39 with the BLOB anti-pattern. This could be explained by the fact that classes, where the BLOB anti-pattern is present and that have a higher coupling between objects, have more method calls to other classes. They both influence IGAM similarly, likely because BLOB and CBO have a high internal correlation of 0.81. To make the calls to other classes, there has to be a higher visibility in the methods of the classes that are being called. This is required by the nature of this type of coding (which could inherently be considered bad coding quality) and causes a higher IGAM value. IGAM is only higher when its methods are more visible then they necessarily have to be. In this case they do need to be more visible, so IGAM has a lower value. This implies the correlation between the higher CBO and BLOB values and the smaller IGAM values.

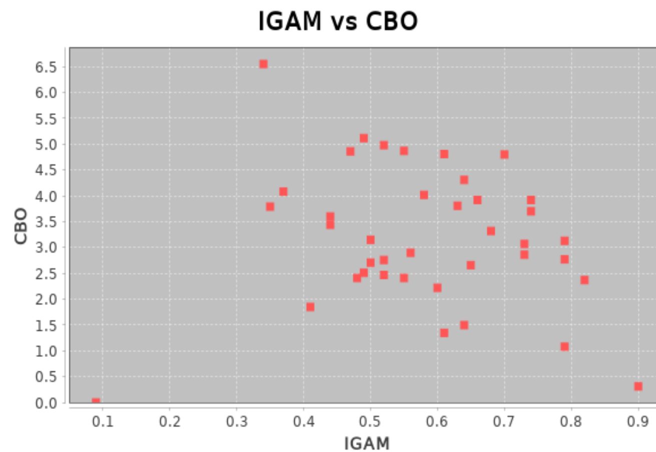


Figure 5.12: IGAM vs CBO

The lack of cohesion in methods also has a relatively high negative correlation with IGAM. Most of the projects that were studied show a higher value of for IGAM at a relatively low value for LCOM5. However, except for some outlying values, the IGAM values do not seem to influence the LCOM5 metric and it mostly fluctuates between 1 and 2. There is a possibility to have very visible, as well as less visible methods in a class, regardless of whether they are cohesive.

### IGAT

Similarly to IGAM, the inappropriate generosity with the accessibility of types metric also has almost no correlations to the quality metrics. We can see four weak negative correlations, of which only one is significant. In the previous study, none of the correlations was significant. The results are very similar, which further adds credibility to the findings.

The only significant correlation for IGAT is with the weighted methods per class metric. It has a value of -0.32. On the graph we can see that some of the most complex projects have the lowest IGAT values and, conversely, some of the projects with the highest type visibility are less complex. I think these few examples are enough to cause the slight negative correlation, but it is very hard to see any kind of relationship between the two metrics in general.

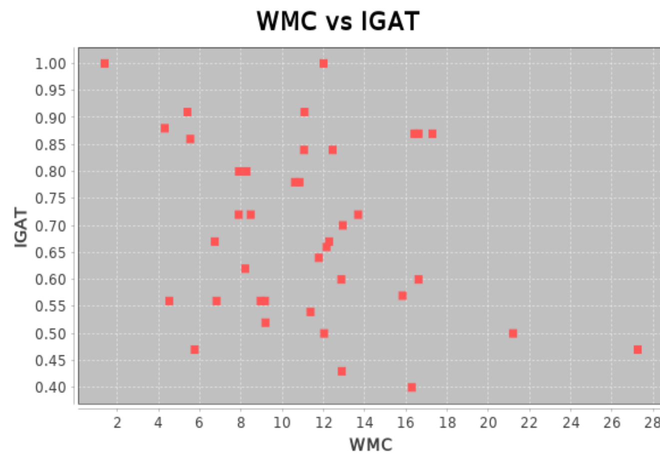


Figure 5.13: IGAT vs WMC

## 5.3 Correlations with the vulnerability metrics

This section is meant for the discussion of the correlations between the vulnerability, quality and security metrics and discuss them. The functionalities of the tool to be able to gather and display such information were implemented, but more data needs to be gathered to be able to discuss any statistically significant correlations between these metrics. The tool is able to gather, store and analyze the vulnerability metrics. However, there are not enough vulnerability metric results to be found for the projects specified in section 6.1 to calculate real correlations. There was only a very low number of projects discovered, that fit all the prerequisites. Even though over 3,000 Gradle + Java GitHub projects were discovered, they did not have many matching vulnerabilities. That is not to say that the NVD does not have vulnerability entries for these projects, but the matching process is technically impaired in a strong way, due to the difficulties described in subsection 4.4.2. It is also likely that this is due to the fact that Gradle + Java projects on GitHub are mostly small android applications. All android applications are built with Gradle, as defined in the android documentation [Goo]. The bigger android projects, which make a lot of revenue, are likely to not be open source and be on GitHub. This means that the metric correlation analysis tool discovered many smaller android applications and those are very unlikely to have a lot of submitted vulnerability data on the NVD.

Due to these circumstances there is not enough relevant vulnerability data to discuss the correlations in question. The study maintains a focus on implementing all the automatic tools for finding more correlations in the future. There are more vulnerability results also shown in chapter 7. For this question, not all projects with vulnerability results could compile, due to issues with some of the other libraries used in the project. Therefore I created two tables to display the kind of results that the metric correlation analysis tool can gather. In Figure 5.15 I show what vulnerability metrics were calcu-

lated by the tool for the projects chosen automatically in chapter 6. I do not show any quality or security metrics in this table, because some of them could not be calculated. In Figure 5.15 I show the vulnerability, quality and security metrics of three projects where all metrics were successfully calculated. This table also shows that the metrics can be calculated for several versions.

Figure 5.14 displays the metric data for the 10 projects that the tool automatically recognizes as vulnerable projects as described in chapter 6. These are the results for which the projects could be matched with complete certainty to their NVD database counterparts. The table lists the project names, the number of unique vulnerabilities discovered for each project, the average CVSS2 score among these vulnerabilities and the maximum CVSS2 and CVSS3 score for them. These are some of the main metrics I defined in section 4.3. Some metrics like the vulnerabilities per 1000 lines of code are missing here, because their values are very small and should also be looked at based on a certain project version. These metrics represent all project versions.

| Product | #Vulnerabilities | AvgCVSS2 | MaxCVSS2 | MaxCVSS3 |
|---|---|---|---|---|
| Smack | 3 | 4.46 | 5.8 | 5.6 |
| Jabref | 1 | 7.5 | 7.5 | 10 |
| Grpc-Java | 4 | 7.5 | 7.5 | 9.8 |
| EthereumJ | 2 | 5.6 | 7.4 | 5.2 |
| Ignite | 4 | 6 | 7.5 | 9.8 |
| Groovy | 3 | 6.6 | 7.5 | 9.8 |
| Kafka | 2 | 5.5 | 5.2 | 6.8 |
| Sonarqube | 1 | 4 | 4 | 4.3 |
| Elasticsearch | 9 | 5.5 | 7.5 | 9.8 |
| Reactor-core | 2 | 4.5 | 5.2 | 5.6 |

Figure 5.14: Vulnerability metrics for all project versions of the discovered projects in chapter 6

Figure 5.15 is a table representing the metric calculation as it is performed for several specific project versions. On the table we can see that the metric correlation analysis tool can calculate all the vulnerability metrics as described in section 4.3. These are highlighted in a light pink color. For a given project, the tool calculates the number of vulnerabilities, the vulnerabilities occuring per 1000 lines of code, the average CVSS2 and CVSS3 scores and the maximum CVSS2, as well as CVSS3 scores. For the project “kafka”, we can see that these calculate for different versions as well, even though the vulnerabilities in question have not been removed. Another example of these calculations follows in chapter 7.

| Application | Vendor | Version | LOCpC | IGAM | LDC | WMC | DIT | LCOM5 | CBO | IGAT | BLOB | LLOC | #Vulnerabilities | VulnerabilitiesPerKLOC | AvgCVSS2 | AvgCVSS3 | MaxCVSS2 | MaxCVSS3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| groupie | lisawray | 2.3 | 38.32 | 0.5 | 0 | 9.16 | 1.02 | 1.37 | 2.71 | 0.56 | 2 | 2414 | 1 | 0.41 | 4.3 | 6.1 | 4.3 | 6.1 |
| kafka | apache | 0.10.2.1 | 47.33 | 0 | 3.4 | 10.07 | 0.85 | 1.38 | 4.79 | 0 | 0 | 100349 | 2 | 0.02 | 5.2 | 6.1 | 5.5 | 6.8 |
| kafka | apache | 0.10.2.0 | 47 | 0 | 3.36 | 10 | 0.86 | 1.38 | 4.77 | 0 | 63 | 100495 | 2 | 0.02 | 5.2 | 6.1 | 5.5 | 6.8 |
| Smack | igniterealtime | 4.0.0 | 46.32 | 0.5 | 5.59 | 10.92 | 0.83 | 1.08 | 4.26 | 0.52 | 26 | 40483 | 2 | 0.049 | 5.4 | 0 | 5.8 | 0 |
| Smack | igniterealtime | 4.1.8 | 40.49 | 0.51 | 3.92 | 9 | 1.59 | 1.05 | 4.47 | 0.52 | 25 | 59433 | 1 | 0.017 | 2.6 | 5.9 | 2.6 | 5.9 |

Figure 5.15: Metric results table with vulnerability metrics

## 5.4 Conclusions

Overall, the number of projects assessed gives good insight about which static code analysis metrics are coherent when trying to assess the quality and security aspects of software projects. The potential of the metric correlation analysis tool to additionally analyze a further number of projects automatically leaves even more promising findings to be expected in the future. Further, the prepared functionality for analyzing vulnerability data correlations also leaves open many possibilities for different future studies.

The results of the correlation analysis show that the given quality metrics are a great set of metrics for the static code analysis of projects. They are very coherent and this can be proven through their many high and medium internal correlations. This was also noted in the previous study and its significance is therefore increased by the reiterated findings. I think that using these metrics, developers can be more confident in the quality of their code. While it is not sure whether that means that less vulnerabilities will appear in their code, a study with looser premises for the sample set could confirm this too. Above all, the biggest projects are related to having worst metrics, which is normal. It is therefore increasingly important to keep an eye on the metric values as developers extend and add functionality to their code. The LLOC metric correlation values show this very clearly. This is also the case for having longer classes and a lot of object coupling. These practices seem to lead to many of the other issues that can be seen from the correlation values for LOCpC, BLOB and CBO. A good advice to derive here, is that it is beneficial for programmers to keep the code complexity lower and pay attention to the separation of concerns. They should avoid creating bigger, multifunctional classes. It is of course generally important to teach developers the principles of quality code design. While some of the aspects of good design, like having smaller inheritance trees, appear less significant, it is still important to look at the quality metrics as a whole, because they have a high internal correlation. There is no single quality aspect that completely outweighs all other aspects. They are all interconnected.

The security metrics chosen for this thesis deliver some more unsatisfying results. I think that IGAM and IGAT are, in fact, security metrics, but they represent a small part of the security spectrum. Their correlations with the quality metrics are negative or insignificant because sometimes having higher IGAM and IGAT values is simply required by the code complexity. The worse the quality gets, the more these inappropriately

generous types and methods are needed. When they are needed, they will not have high metric values, which results in some negative correlations with the quality metrics.

Even though we cannot thoroughly assess the correlation of these security metrics to the vulnerability metrics, there are some things to be said about the aspects of security that they both represent and what is to be expected from their study. Because IGAM and IGAT represent a fraction of the security software they are not likely to be the direct cause of vulnerabilities. They can only be misused if there are already vulnerabilities present. In case there are, and there are generous access modifiers, then these vulnerabilities can be exploited easier. For example, there could be a vulnerability in a game, where an attacker can modify the in-game purchase data sent to the game server. If the attacker wishes to purchase in-game items and he can modify the data being sent about the purchase, he could potentially change his order to receive more items than he pays for. If the method for sending the purchase data has a generous access modifier, this vulnerability becomes easily exploitable. In this case, the modifier itself is not the real vulnerability, but it adds onto it.

I think that the vulnerability metrics represent the security of the projects better. Unfortunately, these metrics cannot be calculated prior to project releases. More security metrics should be found and defined for this purpose. At the time of writing this thesis, the existing options are very limited. However, this was not a goal for this particular study. It is still important to teach developers to limit the access modifiers as much as possible, paying attention to more complex and important structures having the most restricted access. Simpler structures remain mostly harmless regardless of their access restrictions.

The biggest takeaway from this correlation analysis is that quality metrics have a strong interconnection with each other. Making one quality aspect of the projects worse results in a snowball effect on many of the other quality aspects and therefore a negative effect on the whole code base. The principles of good code should be implemented, taught and reiterated to avoid this effect and possibly some costly code vulnerabilities that could come with it. Keeping code small and simple pays off in it being easily extendible. This means developers need less time to implement or change functionalities. Paying attention to this rule will likely show an effect on the occurrence and severity of vulnerabilities and should certainly be studied further. I think that the time spent on improving the software quality is eventually less than the time that would be spent trying hard to maintain the code and mend its vulnerabilities. In that sense, it really pays off to take preventative measures.

# How can the process of selecting and extracting the software projects to evaluate be automated?

This research question is about automating the selection of projects to be used for the metric correlation analysis tool. The previous study, on which this work is based on, was using a manual selection of 50 android applications as the projects to perform its calculations. The goal for this question is to choose more projects to analyze, in order to widen the scope of the present study and to get additional results for the correlation analysis.

## 6.1 Defining features of the projects

To achieve the study goals, some key properties of the sought out projects have to be defined. The arising questions are what kind of projects will not only work best with the tool of the previous study, which is to be extended, but also what kind of projects will generally deliver meaningful results when evaluated against the vulnerability data? The following section lists a few defining features of the projects.

### Open-source projects

The projects have to be open-source. This study will only focus on open-source projects, as other projects are costly and have limited code availability for the users.

### Java projects

This analysis is about the object-oriented quality of code and therefore requires projects in an object-oriented language. Because some of the metric tools can only be applied to the Java language, this is the required language.

### Gradle projects

The projects have to use the build tool Gradle. Gradle is an open-source build automation support system. This is because one of the tools for detection of object oriented code smells and anti-patterns - named "Hulk" - makes use of Gradle. Currently, the tool can only work with Gradle projects.

### Popular projects

Popularity is an important defining feature for the projects, which creates the premise for the metric correlation analysis tool to calculate meaningful results. Popular projects with a lot of contributors or stars are more likely to be considered "real" thriving projects. This boils down to the assumption that any exercise or example projects, that was made for fun, should be left out in order to get an undistorted yield. The usage of any given

"hello world" program, that somebody created in order to learn the basics of coding, should therefore be omitted. Popularity is also important for finding vulnerability entries for these projects in the NVD.

#### Projects with vulnerability data

The projects must have sufficient vulnerability data in the NVD database. Intuitively, the assumption that it is more likely that most projects don't have any data in the NVD because nobody submitted these information, and not because they don't have any vulnerabilities, is highly probabilistic. It is likely that some projects have not undergone enough security analysis and therefore have many hidden vulnerabilities. This could influence the study to classify them as more "secure". To ensure that the results are as meaningful as possible, projects with entries in the NVD should be prioritized.

#### Projects with retrievable version information

In order to match the vulnerabilities from the NVD database to the projects as well as the versions that they belong to on GitHub, there needs to be sufficient information about the version to be easily accessible from their repository.

These are the defined premises for the projects, that the tool for the present study should automatically identify and choose. The next paragraph explains how these premises are applied in the selection process.

## 6.2 Automating the selection process

Using the premises for the projects from section 6.1, the tool creates a database of projects which can be used for the correlation analysis. The following paragraphs explain how each premise is met in this process.

#### Selecting open-source projects

The projects should be freely available for use, so they should be open-source. The project hosting platform GitHub is an easy choice as a source for them, as it is considered a market giant in the area. According to their statistics, GitHub hosted about 67 million software repositories by the end of 2017 [Git]. The tool already supports the use of this platform and the repositories in it, which is another reason to choose it.

#### Selecting Java projects

The GitHub platform has a search API, which can be used to search for the appropriate projects with several parameters, that can be set according to the scope of the search. The extended tool makes use of this API for the selection algorithm. Using the search API provided by GitHub, the tool can easily pick out the Java projects from the other available ones.

### Selecting Gradle projects

A defining attribute of Gradle projects is that they automatize the project build-process. For this purpose, Gradle makes use of a build configuration file called "build.gradle". In this file, dependencies and plugins, which are needed in the application, can be defined. The top-level "build.gradle" file, located in the root project directory, defines build configurations that apply to all modules in a certain project. By default, the top-level build file defines the Gradle repositories and dependencies that are common to all modules in the project. [dev18]

Therefore, to find Gradle projects, the extended tool searches for the existence of a "build.gradle" file in any found Java repositories which are large enough, as described in the next paragraph. Any repositories containing a "build.gradle" file are considered Gradle projects. This process does not require the tool to clone the project, as the GitHub API offers the possibility to search for it on the server side.

### Selecting popular projects

As defined by the prerequisites, the selected projects need to be of a sufficient size, so that their code and vulnerability analysis can deliver large, interpretative results. Measuring their size in terms of byte-size is not a good option, because it is likely that smaller projects can also become widely-used and important. Therefore, this study chooses to look at the project "size" in terms of how popular the project is. This is achieved using the GitHub star system. In GitHub, users can give a "star" to a project that they find interesting, and for which they want to see similar projects in their news feed. Stars are also given to show appreciation to repository maintainers for their work. GitHub uses the star system for many of its repository rankings, as they represent the user satisfaction with these repositories.[Git]

The GitHub search API offers a way to sort the discovered projects by stars. The tool uses this to arrange the discovered projects by given stars and choose projects which have as many stars as possible.

### Selecting projects with vulnerabilities in the NVD

There is no way to use the GitHub search API to discover whether a project has entries in the NVD. Therefore the tool applies all the other prerequisites first and creates a database of suitable projects afterwards. From these projects, a vulnerability search is applied to pick out the ones that do have vulnerability entries. The approach for the tool's search is described in chapter 4.

### Selecting projects with sufficient version information

A practical issue that arises when trying to gather the vulnerabilities of the projects, is that they are associated with a certain project version. However, the GitHub repositories do not clearly state which code belongs to which version of the project. Instead, developers commit their pieces of code to the repository and are not required to annotate it

as belonging to a specific version or initiating a release. There exists the optional choice choice of adding release tags to the commits. A lot of developers do this voluntarily, since it is a form of documenting the software version and the development process. The tool for this study makes use of the GitHub API to retrieve these release tags and to only extract the version number and commit ID from them, documenting the known project versions for each project. They can then be put through the metric analysis.

There are other ways of retrieving the version information, such as scanning some meta files in the project directories for version attributes. In this case, a small study was performed on all the eligible projects for the correlation analysis. The study showed that over 70% of these projects used release tags to document their version information. This study was performed manually.

## 6.3 Evaluation

There are many prerequisites, imposed by the tools structure and context which are mostly not harmful to the final amount of discovered projects. In this case, however, the combination of these prerequisites turns out to be harmful. Finding Java + Gradle projects is simple. The metric correlation analysis tool found more than 3,200 of the most starred Java + Gradle projects for the last ten years and saved them in a database. In order to find them, it searched through more than 9500 total Java projects with over 500 stars. The final selected projects end up having an average of 3911 stars. This means they are relatively popular, and therefore good candidates for further study. One could ask whether this average is sufficient for their popularity. The Java projects with the most stars on GitHub have over 10,000 stars. Such projects, however, are scarce. Because the tool also needs to make sure they fit the other prerequisites, the occurrence is even more seldom. Therefore, 3911 can be considered a sufficient number. Without looking for Gradle-based projects specifically, most of the initial 9500 projects would be eligible for analysis. Adding projects based on the build tool Maven would likely also involve more than half of the 9500 projects.

The limitations begin when the tool rules out the projects that don't have at least one vulnerability. Here are some of the reasons why this could be happening:

- On GitHub there are many helpful libraries, that perform small repetitive tasks, which developers share to help other developers create their programs faster. Many of them posses a lot of stars and because of their simplicity, many of them are not regarded as the kind of projects that have reported vulnerabilities.
- Applications that run on the android platform are all using the Java language and the Gradle build tool by nature. Combined with the fact that GitHub is filled with many helpful projects (which are therefore popular in stars), this means that there are many small android help libraries found by this search. Almost all of these are no big standalone applications and are not likely to have vulnerabilities in the NVD database.

- In relation with the other two points, perhaps stars are not the best indicator of how likely it is that a project will have reported vulnerability data. For future work I think it will be useful to attempt to retrieve the project size instead of the stars, or implement a combination of both. Small, useful tools can have a lot of stars but are not useful for the purpose of this study. The ultimate goal would be to retrieve big projects with a lot of stars. They are likely to have more results.

- A lot of the projects likely did have vulnerabilities, but the problems with matching the projects to their vulnerabilities as described in subsection 4.4.2 remain. This is also an important ground for future work and searching for improvements on the part of the NVD team and some of the GitHub-based project developers.

- The prerequisite of sufficient information on the version of a project is also an issue. Developers are not required to use the release tagging features of GitHub and without this, it is impossible to tell what version of their projects the corresponding commit belongs to. A little more than half of all GitHub projects that I looked at by hand used this feature. I sampled 70 projects, and found that 51 projects used release tags. This is another limitation, that I do not think there is an easy way around. Still, enough projects actually make use of the tagging feature, so that this leads to an acceptable solution.

- Finding projects with release information is not the only problem related to the release prerequisite. Another problem arises with the fact that the release information can be entered in any given way and doesn't need to have a required format. This means that the version information can be found in a string like "1.0" – a good example, or "ProjectX-1.0.0-SNAPSHOT"– a very hard example to work with. The tool has implemented a way to clean up all version information and produce it in a "x.x.x" format. Unfortunately, even with this improvement one can not be sure if this is the way that this version information was entered in the NVD database. The probability of the two not matching is still given.

The combination of these factors leads to a very small amount of matching projects being found in the end. The quantity of those projects aggregate to 10, but not all of them can successfully be built and analyzed for quality metrics by the tool. These projects are:

- "Smack" by vendor "igniterealtime"
- "jabref" by vendor "JabRef"
- "grpc-java" by vendor "grpc"
- "ethereumj" by vendor "ethereum"
- "reactor-core" by vendor "reactor"
- "ignite" by vendor "apache"

- "groovy" by vendor "apache"
- "kafka" by vendor "apache"
- "sonarqube" by vendor "SonarSource"
- "elasticsearch" by vendor "elastic"

To summarize, from the 9,500 projects searched, the projects that fit all mentioned prerequisites except vulnerabilities are a total of 3,200. The tool searches for the vulnerabilities of each project in its local vulnerability database. Any project with at least one vulnerability is selected for the metric correlation analysis. The number of projects with at least one vulnerability add up to the sufficient result of 10. These projects are the input for the present correlation analysis study.

## 6.4 Conclusion

Using the GitHub API, it is relatively easy to discover projects of a specific language and a specific popularity. It is harder to discover which projects use Gradle as a build-tool, because additional requests have to be sent to the GitHub API about each individual projects. These requests require the API to search the project structure and also require extra computation time. The GitHub API also limits the amount of results that queries can retrieve and the amount of queries per minute greatly, making the selection process additionally slower.

When discussing the vulnerabilities for the projects, it is better to be able to discover more. There are some limitations to how many can actually be discovered, and ideas about future work against those limitations can be found in chapter 10. For example, their number could likely be increased by including projects that also use other build-tools such as "Maven". It is also not to be neglected that the names and vendors of the projects from GitHub can sometimes hardly be related to the names and vendors of the projects according to the NVD. More consistency between the two resources would allow a higher number of projects to be studied. Other limitations like these are discussed in subsection 4.4.2.

Additionally, the need for proper version information gathering is not a limiting option that can easily be dropped. It would be a good idea to retrieve more projects by first broadening some of the other requirements. The developers on GitHub must enter their version data in some way for the tool to discover it. They are not strictly required to do so, so if they choose not to, it is impossible for the metric correlation analysis tool to study their projects.

Having mentioned all the limitations that this research question was faced with, there are also many positive changes that were implemented in the metric correlation analysis tool. For one, it now has an automatic way of selecting projects for the analysis and

it can easily be extended with different prerequisites for these projects with the simple interfaces that it offers. Also, it automatically creates a clean output file with all the projects and their version information, entered in a uniform way. There are many definite proposals for future work with the now automatized tool that only needs to deal with some of the very specific issues related to the vulnerability matching. It now includes an overall powerful, extendable as well as reliable GitHub crawler.

# How does the correlation between the quality, security and vulnerability metrics behave across multiple software versions?

In chapter 5, I discussed the correlations between the quality, security and vulnerability metrics of the project in terms of a single project version. In the present research question, I study the evolution of these metric correlations across several versions of a specific software project. The purpose of this research question is to see how the development of the metrics takes place and if certain trends in their evolution can be discovered.

For this research question, I extended the metric correlation analysis tool with more automatic analysis functionalities. The tool is now able to take all the metric information for the projects with its respective version and generate graphical representations of their development. Even with a smaller amount of vulnerability data, as described in subsection 4.4.2 and section 7.3 the future possibilities here are also very promising. As soon as looser project prerequisites are defined and therefore more projects can be analyzed, their vulnerability data can easily be represented graphically with the present functionality of the tool. To prove that this is possible, two examples of projects with vulnerability data are also shown in this chapter.

In the next section, I go over the trends for six of the projects with the most version information on GitHub, which also successfully compiled for the correlation analysis by the tool. I show their graphical representations and discuss possible reasons for their developments in terms of the quality and security aspects. After that I go over the projects with vulnerability data statistics.

## 7.1 Metric evolution trends of the quality and security metrics

To discuss the metric evolution trends for this research question, I use graphics generated by the tool. The X-axis of the graphics display the version changes, e.g. “version 1.0 -> version 2.0”, and on the Y-axis the percentual metric changes in the values of their metrics for the switch from one version the next are shown. The different metrics are represented in different colors, which are stated in the legend on the bottom of each graphic.

### 7.1.1 Metric evolution trends

#### Metric evolution trends for project Barcodescanner

The first project that I will discuss the metric evolution trends for is “barcodescanner” by the vendor “dm77”. Barcodescanner is an Android library for scanning bar codes with 4,458 stars on GitHub as of February 2019. It is therefore a relevant and popular Java project to look at. The metric trend graph for this project is represented on Figure 7.1. On the graph we can see that the WMC and LOCpC metrics have an extremely similar

development across all versions. If we look back at chapter 5 this confirms the 0.91 strong positive correlation between these two values very confidently. This is a result that had been discovered in the previous study as well. Moreover, the LLOC metric perfectly mimics the development of the LOCpC metric on this graph. The line for LLOC completely covers the LOCpC line. They have a medium correlation of 0.52 and this might mean that for this project the lines of code were evenly distributed across several classes, so the average and the total increased similarly. On the other hand, the CBO metric presents an opposing development to these two metrics for all versions. This is not an expected result, as the correlation values in chapter 5 are above 0.45 for both of these pairs. It is also interesting to notice that the lowered duplicate lines of code led to most of the other metrics increasing in one of the versions, but had a similar development in later versions. My suggestion would be that the reduction of the duplicated code was as a result of new code being written in order to replace the old duplicate one, directly increasing the LOCpC value and therefore the complexity. The complexity and lines of code became slightly lower towards the middle of the graph, seemingly because the classes became more codependent (higher CBO value). Finally, more code was duplicated, perhaps including more complex methods, which again raised the LOCpC and WMC metrics. Because the code was then duplicated, the CBO didn't necessarily increase.

Some of the metrics for this graph have not had any changes within these versions and are therefore not displayed. For example, this includes IGAM and IGAT. They are presented in some of the other graphs.

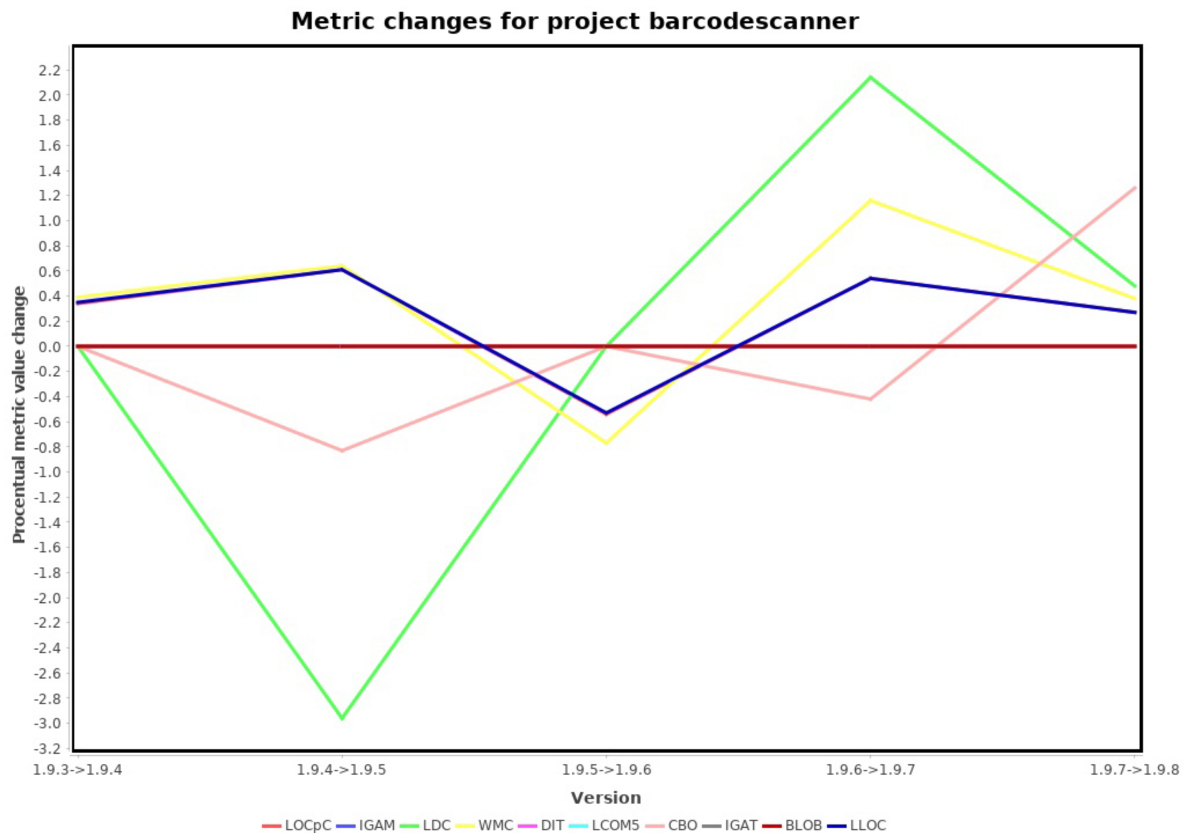


Figure 7.1: The metric developments for the project “barcodescanner” by the vendor “dm77”

### Metric evolution trends for project GraphView

The second project that I will discuss the metric evolution trends for is ‘GraphView” by the vendor “jjoe64”. GraphView is an Android library for programmatically creating flexible and stylized diagrams with 2,268 stars on GitHub as of February 2019. The metric trend graph for this project is represented on Figure 7.2. We can see much more similar developments on this graph as opposed to the one for Barcodescanner. The duplicate lines of code metric seems to fluctuate between similar and opposing development, giving the notion that it really depends on what kind of code got duplicated. If good code, without much coupling, is duplicated, the other metrics won’t necessarily rise as much. If bad code is duplicated, the other metrics will clearly rise as well as an effect of the duplicated code. This may be an indication that the LDC metric should be viewed in combination with other ones and therefore be given a slightly lower importance than for example the WMC metric. Another thing we can tell from looking at this graph is the slight negative evolution of the IGAM and IGAT metrics compared to all the quality metrics. This further confirms the results of the second research question discussion in chapter 5. One thing that really sticks out is the evolution of the BLOB anti-pattern

metric. In one of the versions, it has an opposite evolution to the other quality metrics. If we look back at the correlation results, the BLOB metric had only weak correlations to LDC, WMC and DIT. The LDC and WMC metrics are furthest from the BLOB metric in this version and this is a good indication of the aforementioned correlation results' validity. We can see that the LDC metric can be high or low independently of BLOB, because they correlate with a value of 0.4. In general, all quality metrics follow a similar pattern other than one specific version. The IGAM and IGAT metrics mostly show small opposing trends to the quality metrics as expected from their correlation values.

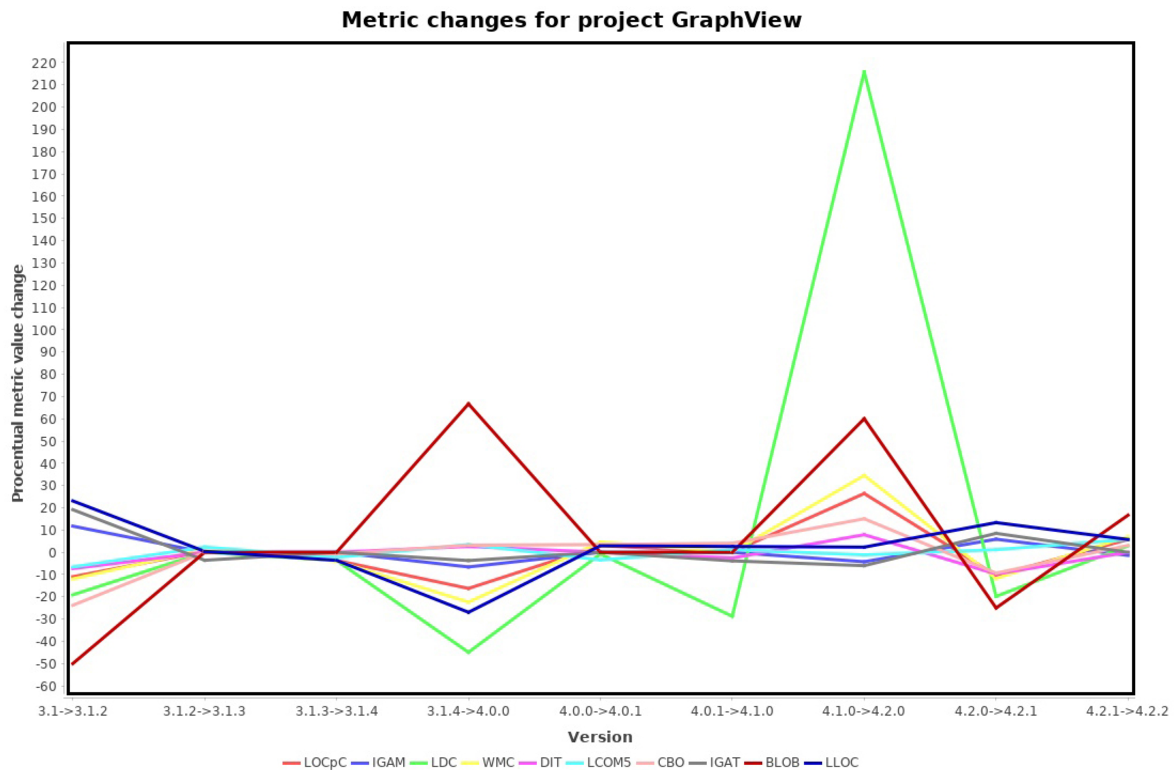


Figure 7.2: The metric developments for the project "GraphView" by the vendor "jjoe64"

### Metric evolution trends for project Gson

The third project that I will discuss the metric evolution trends for is "gson" by the vendor "google". Gson is a Java serialization and deserialization library to convert Java Objects into JSON and back. It has 14,767 stars on GitHub as of February 2019. It is one of the most popular and widely-used Java project to look at, which makes it a very interesting project to analyze the metric evolution for. It was also used in the implementation of the metric correlation analysis tool. The metric trend graph for this project is represented on Figure 7.3. A high amount of versions was analyzed for this project. For them, we can definitely see that the LDC values follow the other quality

metrics more closely than in the previous two projects. The WMC and LOCpC metrics confirm their strong correlation here too. One interesting development is that of the DIT metric. It has a very low correlation with LOCpC and therefore they do not follow a similar pattern most of the time. Because WMC is very closely related to LOCpC, the DIT metric diverges from it too. DIT also shows a clear opposing development to the IGAT metric. This can be explained by their weak negative correlation of 0.31. The IGAM metric fluctuates between opposing and similar development compared to the quality metrics, with similar patterns for most versions. This is an unexpected result, but it is still possible, since IGAM has only one medium negative correlation, which is with LLOC. This correlation can be seen for some of the versions, but it is not so prevalent later on in the project development.

It is interesting to see that the size of the project in terms of LLOC doesn't increase steadily, but instead goes up and down from version to version, remaining steady for some time as well. While the LLOC is steady none of the other metrics fluctuate. This might be due to several versions introducing some metadata or similar irrelevant changes. As soon as LLOC changes all other metrics start changing as well. It is also interesting to see such stark changes in the LDC values. It looks like those changes influence the other metrics slightly, but not too dramatically, since it in fact depends on what kind of code was duplicated. LDC has medium or weak correlations with all but one quality metric.

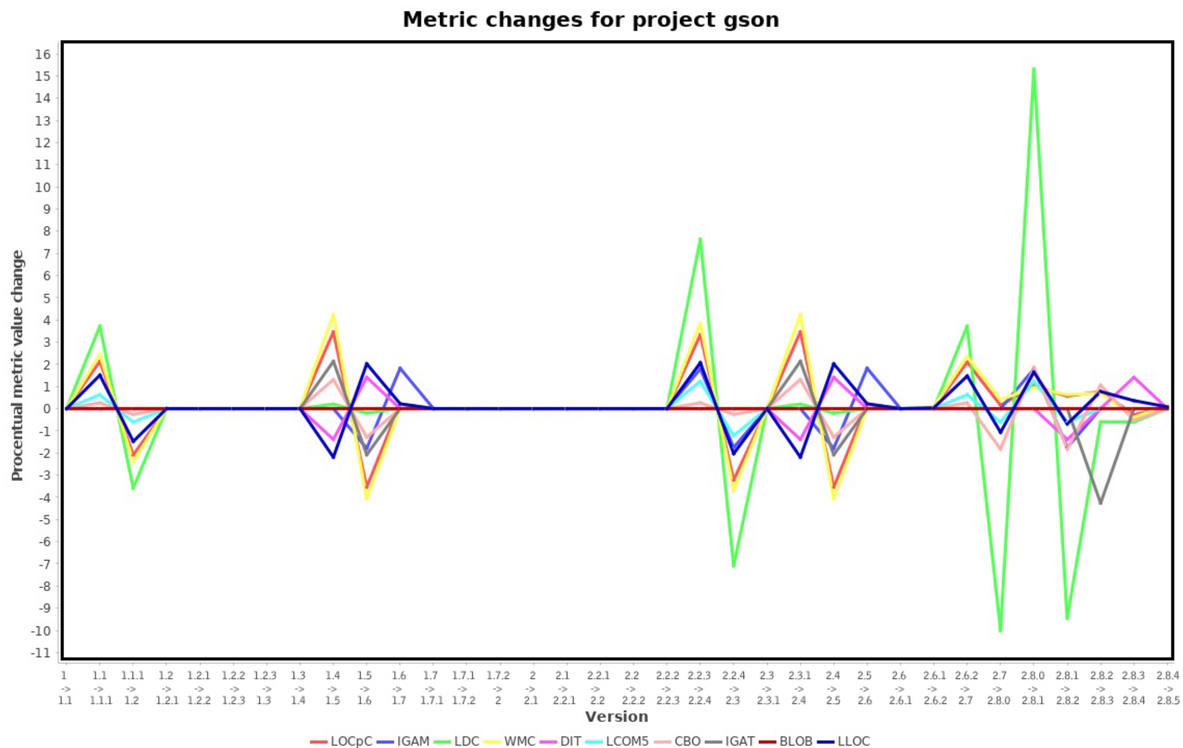


Figure 7.3: The metric developments for the project “gson” by the vendor “google”

## Metric evolution trends for project JsonPath

The fourth project that I will discuss the metric evolution trends for is “JsonPath” by the vendor “json-path”. JsonPath is a Java library for querying JSON files. It has 3,465 stars on GitHub as of February 2019. The metric trend graph for this project is represented on Figure 7.4. The first thing to notice on this graph is the very distinct metric development of all metrics. They have strong fluctuations with sharp peaks. The LOCpC and WMC metrics have a nearly identical development as in all the previous graphs. The IGAM metric follows very similar patterns as the IGAT metric throughout all versions. According to the correlation table, they only correlate with a weak value of 0.35. This development is not always a phenomenon, if we go by the previous graphs and it is likely that one cannot make clear assumptions about one of these metrics by knowing the value of the other. The IGAM metric has a clear opposing development to the LOCpC and WMC metrics, DIT and CBO. Conversely IGAM seems to have some similar developments as the LLOC and LDC metrics. I think this might be due to an increase in duplicate code, and an LLOC increase in general, which led to many inappropriate access methods and types being duplicated. The LDC metric fluctuates between very high and relatively low values and seems to have only small influence on the other metrics. The DIT metric follows the LOCpC and WMC metrics closely in this graph, even though they do not show any significant correlations with each other. This is not a

phenomenon in the other graphs and appears to be a random occurrence, confirming the correlation results from the second research question. The LCOM5 metric has mostly similar developments to the other quality metrics, but seems relatively independent of LDC and the BLOB anti-pattern. It has a near medium correlation with both of these metrics, but this is not very present in this specific project. The BLOB anti-pattern metric seems to oppose the development of the LDC metric and show a relatively similar development to the LLOC metric for most versions, as expected from their correlation having the value of 0.94.

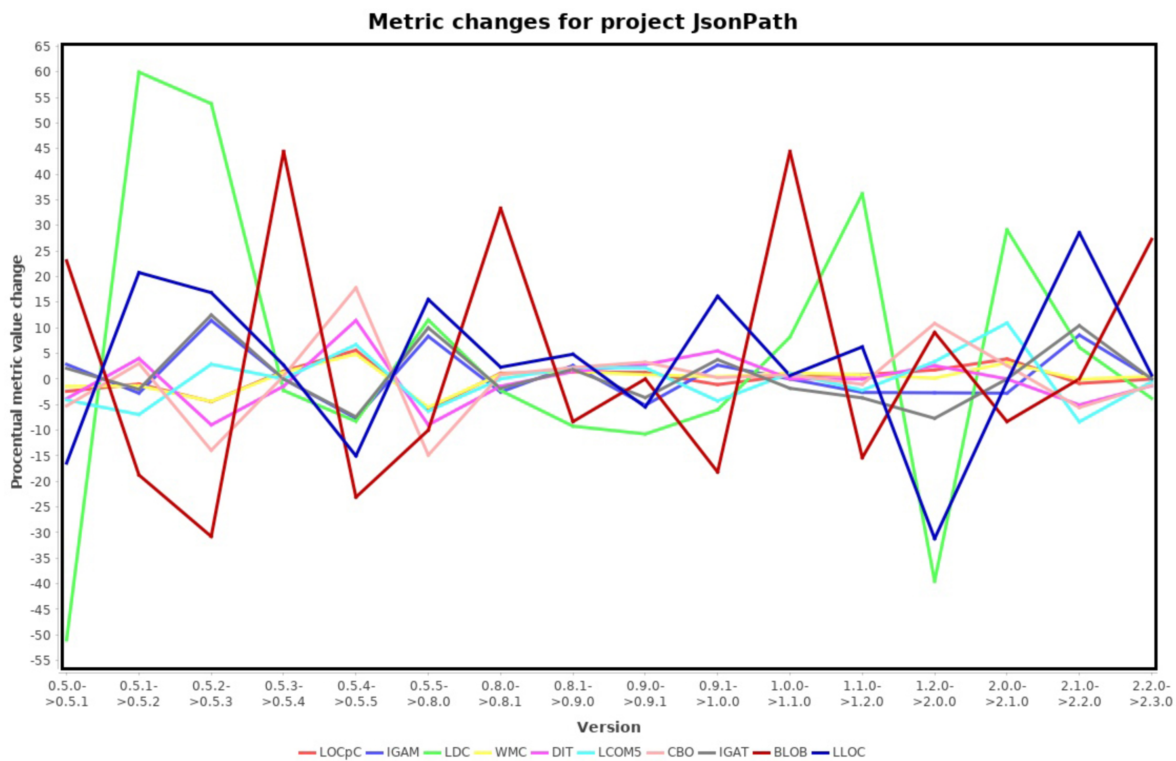


Figure 7.4: The metric developments for the project “JsonPath” by the vendor “json-path”

## Metric evolution trends for project Malmo

The fifth project that I will discuss the metric evolution trends for is “malmo” by the vendor “Microsoft”. The project Malmo is a platform for Artificial Intelligence experimentation and research, built on top of the game Minecraft. It has 3,144 stars on GitHub as of February 2019. The metric trend graph for this project is represented on Figure 7.5. The metrics on this graph have slightly more ordinary developments than the previous one. As in all other graphs, LOCpC and WMC are very close together. They have a similar development to LCOM5 and CBO as well. The development of the LDC metric seems to be sporadic again, and sometimes relates to the other quality metrics, but other times seems completely random without any dependency. The IGAM

metric mostly opposes the quality metrics, as predicted by their negative correlations with each other. We can see the IGAM decrease whenever there are positive increases in the quality metrics. LDC does not seem to influence it. Even though DIT and CBO have a high positive correlation of 0.73, this is not the case for all versions on the graph. Their positive correlation is very visible towards the final versions.

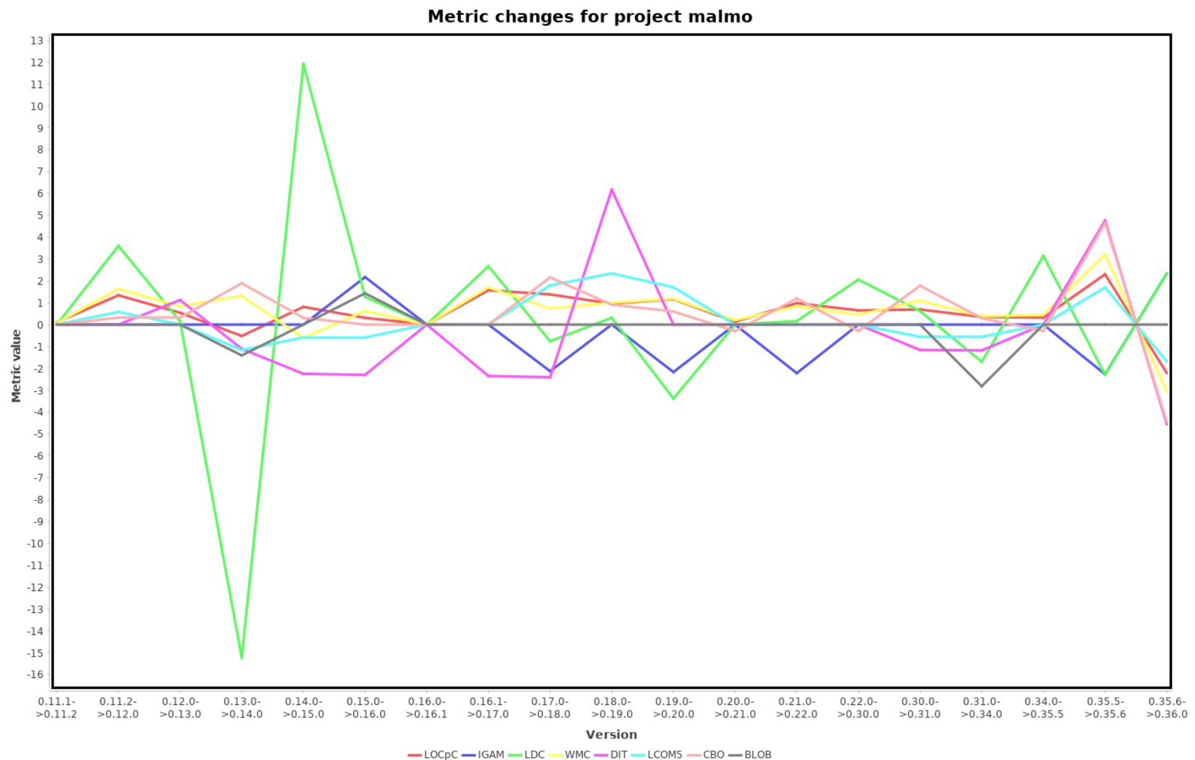


Figure 7.5: The metric developments for the project “malmo” by the vendor “Microsoft”

### 7.1.2 Metric evolution trends for the project Smack

The sixth project that I will discuss the metric evolution trends for is “Smack” by the vendor “IgniteRealtime”. The project Smack is a modular and portable open-source XMPP client library written in Java for Android and Java (SE) virtual machines. It has 1,811 stars on GitHub as of February 2019. At a first glance on this graph we can see the LDC metric has stark upward and downward tendencies and therefore fluctuates a lot. As in the other graphs, this does not seem to affect the other quality metrics dramatically. The LOCpC and WMC metric have a very similar evolution, and seem to correlate negatively with the DIT metric. Their correlations are not negative according to the correlation table in chapter 5, but they are inconclusive. If we judge by the trends on all six graphs, it really depends on the project whether these metrics will correlate positively or negatively, making any conclusions impossible. The IGAM metric fluctuates without showing any particular tendency towards the quality metrics, remaining slightly

negative where it does as expected. The LCOM5 metric follows the other quality metrics more closely, as in most of the other other graphs, while the CBO metric only does this for some of the versions. The CBO metric has relatively strong positive correlations with the other quality metrics, especially with DIT (0.73) and BLOB (0.81). It seems to be the influence of the strongly fluctuating DIT value between version 3.4.1 and 4.1.0 that moves the CBO metric values away from most of the other quality metrics. As the DIT metric stabilizes, the CBO metric returns to a close correlation with all of them, especially with the CBO metric. For this reason, I think it is important to look at the quality metrics as a whole and not individually, as they are very interconnected. The stronger correlations between some of them can create the impression that a certain metric is behaving unexpectedly at times. Therefore, conclusions should be made only by looking at the quality metrics as a whole. A complete conclusion subsection, where I will discuss about the findings and their coherence, follows later in this chapter.

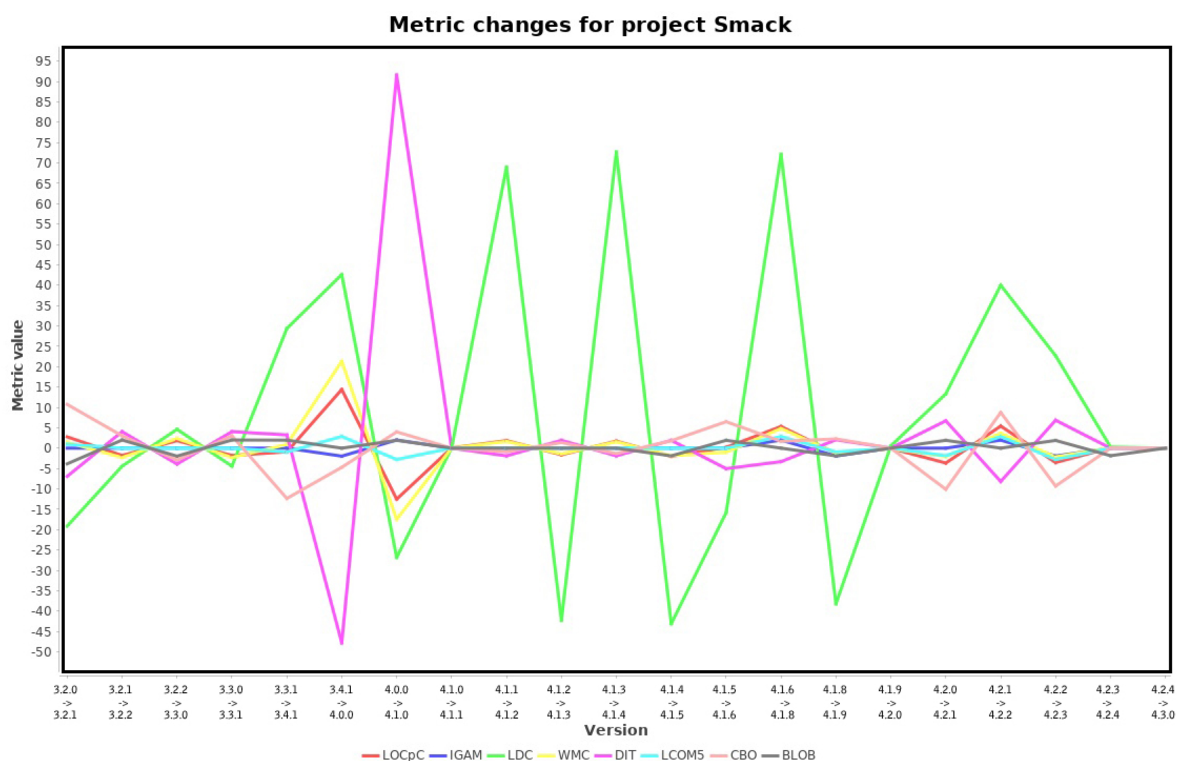


Figure 7.6: The metric developments for the project “Smack” by the vendor “IgniteRealtime”

## 7.2 Metric evolution trends of the vulnerability metrics

In this section we study how the vulnerabilities change, using the example of Smack as a reference, and discuss their relationship with the quality and security metric trends from the previous subsection. This discussion will display the possibilities for the tool

to analyze metric trends for vulnerabilities across several versions as well. The values of the Y-axis for this graph do not represent percentual changes. The reason for this is that the vulnerability metrics can have a valid result of 0 for some versions. Therefore, the Y-axis values on the vulnerability metric evolution graph represent the absolute changes in the vulnerability metric values.

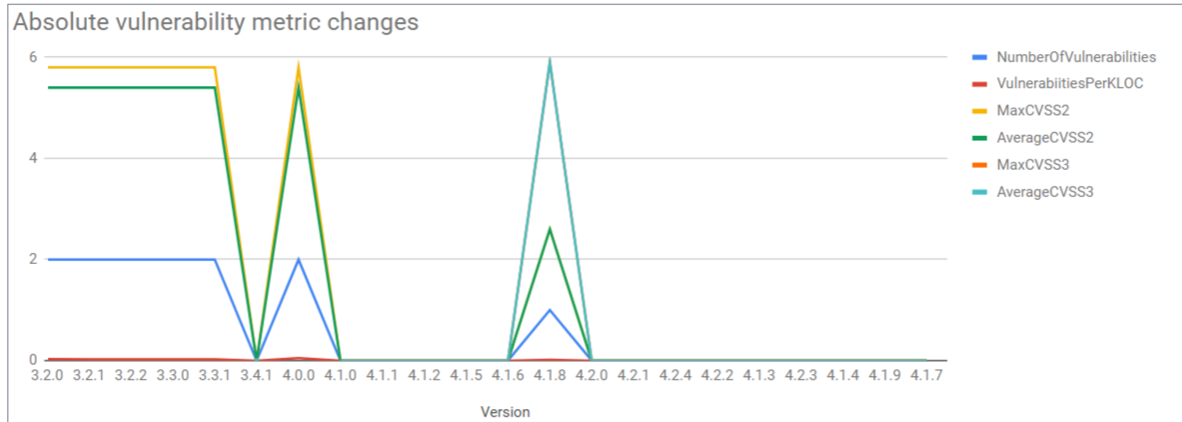


Figure 7.7: The vulnerability metric developments for the project "Smack" by the vendor "IgniteRealtime"

On Figure 7.7 we see the evolution of the number of vulnerabilities, vulnerabilities per 1000 lines of code, average CVSS2 and CVSS3 score and maximum CVSS2 and CVSS3 score metrics. The graph starts with two vulnerabilities, that disappear in version 3.4.1 and reappear for one version right after this. In the quality and security metrics graph for this project we can see a relatively strong decrease of the CBO metric and a relatively high increase of the LDC metric for version 3.3.1 -> 3.4.1. Most of the other metrics maintain a steady level at this point. This version change is where the vulnerabilities are resolved. It is possible that some of the code was replaced with duplicate code, removing some classes or libraries that had a high coupling. This likely resolved the vulnerabilities, but can only be considered a shorter-term solution.

In the following version change, where the vulnerabilities reappear, there are several more obvious metric changes. Namely, the DIT metric decreases dramatically, the LOCpC and WMC metrics increase, and the LDC metric continues to climb. There is also a slight increase of the LCOM5 metric. This version change is major, so much more code was written, which ended up increasing the code base both with duplicated and complex additions. The inheritance tree likely shortened due to some classes being merged together (thus the LOCpC increase). This seems to have brought back the vulnerabilities, and required another fix. In the local elasticsearch database I found that at this point the vulnerabilities are still the same. The one that appears later on is a different submission.

The version change from 4.0.0 to 4.1.0 resolves these vulnerabilities completely. On the quality and security metric graph for this version change we can see that a lot of the duplicate code was deleted, including a lot of complex code. The average class size decreased as well. The coupling between objects increased, likely due to the smaller classes requiring more interconnection between each other. The most prominent change is the stark increase of the DIT metric. A lot of structure seems to have been introduced into the code, while the general repetitiveness and size decrease simplified it. It is also possible that most of the changes from the previous version were reverted and this caused most of the metrics to show an opposing development. The newly introduced code simplicity must have helped the developers to resolve the vulnerabilities more efficiently. The DIT metric has the most dramatic changes between the versions where the vulnerabilities reappear, but it still needs to be viewed within the context of the other metrics.

In version 4.1.8, another vulnerability appears. The quality metrics are increased only very slightly in this version change, with only the LDC metric steadily decreasing. The CBO and DIT metrics behave differently around the appearance of this vulnerability, which means it might be related to another metric change. The only metric that changes significantly in this version is the LDC metric. Perhaps removing some of the duplicate code, also slightly increased the object coupling and created an opportunity for a vulnerability to appear. In the next version change, some of the duplicate code is removed, which increases the WMC, LOCpC and CBO metrics and even introduces some anti-patterns. One similarity, to when the previous vulnerabilities were resolved, is the increase in the CBO metric. While it might support bad style, it looks like sacrificing style might sometimes resolve vulnerabilities. This could make the code base easier to understand for more developers. As long as it does not come with an increase in WMC, it could indicate that the CBO metric increase has something to do with vulnerability removal. The IGAM metric does not show any significant relationship to the vulnerabilities, in fact it decreases slightly for versions 4.1.6 to 4.1.8 and increases again, when the vulnerabilities are resolved in version change 4.1.8 to 4.2.0. It is hard to say what metric influenced the vulnerabilities to appear from this amount of data alone, but the possibilities for future analysis are very promising in this regard.

If we look at the severity of the vulnerabilities appearing in 4.0.0 and 4.1.8, the latter ones are less severe in their CVSS2 score. The CVSS3 score classifies the 4.1.8 vulnerability as very severe, but was not entered for the ones before that. Judging by the stronger fluctuations of the quality metrics around the occurrence of the first two vulnerabilities, it looks like the more dramatic quality metric changes cause worse vulnerabilities to appear. We cannot be sure what this comparison would look like if there was a CVSS3 score for all vulnerabilities as well. Unfortunately this scoring system appeared later in the NVD development, and vulnerabilities that were entered before its creation simply do not contain the CVSS3 score value for the period prior of its introduction.

## 7.3 Conclusions

Analyzing the metric evolution trends was definitely an important part of the metric correlation analysis. Much more additional insight about the correlations is gained when looking at the trend graphs. Some conclusions can be drawn, which are probably impossible to make from the correlation tables alone. These graphs give the opportunity to see which metrics tend to have a similar or opposing evolution and which metrics tend to seemingly fluctuate regardless of the other ones. A variety of additional trend graphs can easily be generated in the future to see if these trends will remain similar.

For the six graphs that I analyzed, most of the trends do remain very similar. For example, every graph showed a strong relationship between the trends for the LOCpC and WMC metrics. The IGAM and IGAT metrics also showed their mostly negative correlation with the quality metrics in most graphs. Both of these developments were expected from the correlation table in chapter 5. The LDC metric was fluctuating between positive and negative with the strongest changes and always had relatively small effects on the other metrics changing. The DIT metric behaved in a mostly unexpected way for most of the graphs and it would be interesting to see what conclusions can be made about it with more data. The LCOM5 and CBO metrics tended to mimic the development of most of the other quality metrics, usually WMC and LOCpC. They prove their correlations to the other metrics, as discussed in chapter 5. The BLOB metric was behaving more independently, compared to the other quality metrics despite its significant correlations with them.

I think it is best to look at the trend graphs as a whole, considering all metrics displayed on them. There is no doubt that the quality metrics influence each other, but there are some specific pairs of metrics that can stand out from the rest. The specific circumstances can likely be deduced from studying the graphs more closely, as I have done for each separate analysis. The information gained out of this graphs will be especially useful for the developers of these projects, who might have some more insight of what specific circumstances might have caused the metrics to behave unexpectedly. With this knowledge, they could likely derive much information from the graphs and focus on the worst metric results, to lower some of the other ones that depend on them. They can also learn from past mistakes, remembering what caused sudden increases in all metrics. For example, if the LDC metric caused a lot of their other metrics to worsen in the past, maybe they should look at the qualities of the duplicate code in their project. If the LOCpC caused an increase in the BLOB and WMC metric, maybe they should focus on the separation of concerns in their project. They could also try to lower the negative correlations with the IGAM and IGAT metrics, as these symbolize the necessity for higher method and type accessibility due to bad quality. A smaller negative correlation could be a good sign for their quality metric values.

Using the vulnerability metric graphs will likely offer an opportunity to assume what quality and security metrics should be first worked on when low vulnerabilities are crucial in a project. I think it is very likely that some of the quality metrics from this study have significant correlations with the vulnerability metrics and hope this can be proven in the future. To this end, users and developers should try to report more vulnerability data to the NVD and other resources to give this method better chances of success. Perhaps they could also gather their vulnerability data internally and find other ways to evaluate it.

# Implementation

In this chapter I will explain how the metric correlation analysis tool is implemented, focusing on some of the new aspects from the present study. To this end I have created Figure 8.1 which should help the reader to better visualize and aid the explanations in the following subsections.

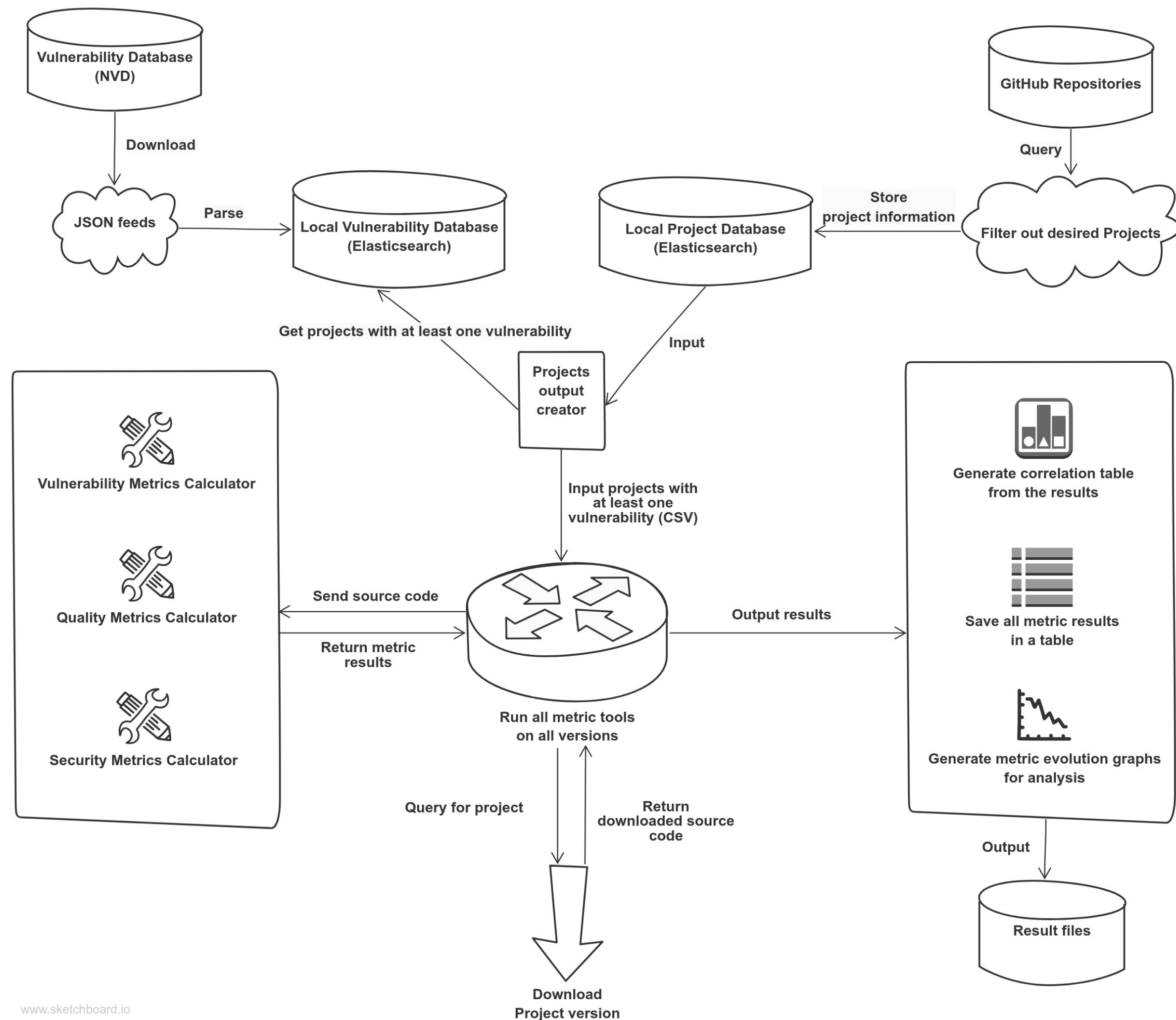


Figure 8.1: Implementation graphic for the present study

The overall flow works as follows: for the first research question, the tool retrieves the vulnerability data from the national vulnerability database in the form of JSON feeds. It parses the retrieved information and stores it as entries in a local Elasticsearch database (described in section 8.1). This is the top left part of the graphic. For the

third research question, the tool queries the GitHub repositories, given certain prerequisites from chapter 6. The retrieved project information from GitHub is also stored in a local Elasticsearch database. Then the tool searches for vulnerabilities in all the locally stored projects that fit the prerequisites. If the tool finds that a project has at least one vulnerability, it retrieves the project's version and commit information and stores it in a CSV file. This is done in the *ProjectsOutputCreator* class. Using this CSV file with all the project, version and commit information, the tool can proceed to download every project version and run the metric analysis on it. This is represented on the bottom center of the graphic. With the project's versions downloaded, the tool runs all metric analyses for every project by inputting its source code. The result is a map of metric data. This is represented in the rectangle on the left side of the graphic. Finally, given the metric map, the tool can output different results. It outputs correlation tables for the second research question, a metric result table and metric evolution graphs for the fourth research question. This is represented on the right side of the graphic. These files are the definite output of the metric correlation analysis tool.

In the following sections I will go over the different parts of this graphic in a little more detail, giving method and class names. I will also present the Elasticsearch database.

## 8.1 The Elasticsearch database

The choice of this specific database rests upon a few considerations: To begin with, relational databases work relatively slow and less efficient when fetching search results from large amounts of data, compared to NoSQL databases [NPP13]. NoSQL databases are essentially databases that find a way around using the structured query language SQL to retrieve data. Because the vulnerability data from the NVD includes more than 100,000 vulnerabilities, a NoSQL database is a better choice for quick data retrieval. Another reason for this choice is that Elasticsearch internally deals with its data in JSON form, therefore it is easier to import and handle the initial data from the NVD. Finally, the main reason for the Elasticsearch database utilization is the possibility for fuzzy searching the entries. This is a very important requirement for the second research question and is used on several different occasions in this project.

## 8.2 Retrieving and storing the vulnerability data

All files that provide the core functionality for retrieving and storing the vulnerability data are found in the *metric.correlation.analysis.vulnerabilities* package. To download and parse the JSON feeds from the NVD database locally, one has to create an instance of the *VulnerabilityDataImporter* class. This class essentially checks the NVD database web page for updates, downloads and unzips them and then parses them using the Google "gson" library. To parse them, it deserializes them into objects of the class *Vulnerability*. It then stores these objects in a local Elasticsearch database, but only includes them with their relevant information as described in section 4.2. The method *parseJSONFilesToDocuments();* can also be edited to retrieve other vulnerability data.

Storing the vulnerability data locally means that the tool can easily query for it, since this is not offered by the NVD. The local database name is defined in the *vulnerabilityDatabaseName* global variable.

## 8.3 Automating the project retrieval process

All files that provide the core functionality for the project retrieval automation are found in the *metric.correlation.analysis.projectSelection* package. To discover projects that fit the prerequisites defined in section 6.1, the tool utilizes the GitHub API. The API offers several query functionalities and comes with some limitations. In the method *searchForJavaRepositoryNames();* of the class *ProjectSelector*, the tool utilizes 30 requests per minute for a given number of projects that should be analyzed. The first request made by the tool is aiming for Java projects. For each Java project found as the result, the tool then then applies the other filters. Currently the only filter is the Gradle build nature filter in the *GradleGithubProjectSelector* class. It extends the *IGithubProjectSelector* interface, by which other filters can be added. If a project passes all filters, it is added to a local Elasticsearch database (name defined by the *repositoryDatabaseName* variable in the class) in the method *addDocumentsToElastic();*. The information stored about every project consists of its name and its vendors name.

One of the limitations to look out for here is that GitHub only allows the 30 requests per minute for GitHub users with a valid OAuth token. The token string should be replaced in the variable *oAuthToken*. Even with 30 requests per minute, the GitHub API sets another restriction of 5000 requests per account per hour. This means that to analyze further projects either more accounts should be used or a pause should be made every 5000 requests. For the present study the pause was induced manually.

## 8.4 Retrieving projects with at least one vulnerability

The following functionalities are found in the *metric.correlation.analysis.projectSelection* package. To locate the projects that have at least one vulnerability, the tool retrieves all the projects stored in the local projects database and queries the local vulnerability database for any entries with their product and vendor name. This is done by the method *getProjectsWithAtLeastOneVulnerability();* in the *ProjectSelector* class. The class *ProjectsOutputCreator* uses the retrieved projects from this method to query the GitHub API once more for any version and commit information of the projects with vulnerabilities. This additional information is required for the second and fourth research questions. To download a specific project version from GitHub, the tool needs to know the ID of the commit related to the version release. In the *ProjectsOutputCreator* class all version and commit information is gathered for the relevant projects with vulnerabilities. This information, the project vendor and project name are all stored in a local CSV file (name defined by the variable *projectsDataOutputFilePath*). To generate this file, the user has to call the *getProjectReleases();* method.

A thing to look out for here is that the GitHub project version information is added in various ways by the developers. A project version could be denoted, for example, as “1.1-RELEASE” or “gson-version-1.1” when retrieved from GitHub. This is why another method is required, which takes the outputted file and normalizes all version information inside of it (it would change the version to just “1.1”). This method is called *cleanUpProjectVersions();* and creates a normalized, ready to use CSV file with projects and their version and commit information (name defined by the *normalizedProjectsDataOutputFilePath* variable).

## 8.5 Retrieving the project metric data

The functionality for downloading a project from GitHub and building it locally was part of the tool since the previous study so it will be explained in less detail here. Using the normalized project output, the tool can generate an instance of the class *ProjectConfiguration* which includes all the relevant data for downloading. This data is utilized by the class *GitTools* which has two methods - a method to clone a GitHub project (*gitClone();*) and a method to pull the next version of the same project by only downloading the changes since the last commit (*changeVersion();*). These methods are used in the *MetricCalculation* class with an input of a given *ProjectConfiguration*.

For every downloaded version, the tool needs to calculate all metric values from all tools and store them. The metric calculation tools are initialized in the constructor of the *MetricCalculation* class. There, they can be easily extended with new tools if they are packed into classes that implement the *IMetricCalculator* interface. All metric tool definition classes can be found in the *metric.correlation.analysis.calculation.impl;* package. The method *calculate();* of the *MetricCalculation* class gets the source code of a given *ProjectConfiguration* and runs all metric calculation tools on it as initialized in the *MetricCalculation* constructor. The results of this calculations are saved in a local CSV file in the “results” directory. For the fourth research question (getting data for several versions), the method *calculateAll();* is used. Additionally, all metric result values are saved in a LinkedHashMap called *allMetricResults*. This hashmap is used for the statistical calculations.

## 8.6 Generating the metric analysis output

Given the metric results hashmap, the tool can generate different outputs for analysis. For one, it saves the raw results as a CSV file containing all possible information about the projects and their metrics. This file is generated in the “results” folder. This is done by initializing the *MetricCalculation* class. The statistical outputs are realized in the *metric.correlation.analysis.statistic* package.

The method *performStatistics();* in *MetricCalculation* initiates the statistic generation for the second research question. It is done by the method *calculateStatistics();* in the *StatisticExecuter* class. This method takes the metric results and a desired output file

and writes the CSV results of the Spearman correlation calculation to it. The Spearman correlation matrix is calculated using the Apache correlation calculation class from their library. Besides outputting the correlation matrix to a file, this method also uses the "jFreeChart" library to generate the graphical representations of the correlation results, paired for each combination of metrics. This is all the needed output for the second research question analysis part.

By using the *calculateAll();* method for the metric results generation, the data for many project versions is stored in the main project results file. This data can be used to generate the metric evolution graphs for the research question four analysis. The result needs to be moved manually to the "input" folder and renamed to "versions-results.csv". This can be automated in the future. Once the file is there, the method *writeVersionsCSVFile();* can transform it into an input file for graph generation by taking the metric values and turning them into change values based on the project versions. Essentially, this means that if we have the project "elasticsearch" with an LLOC metric value of 1 for version 1.0 and 1.5 for version 2.0, the tool will turn this information into "elasticsearch has an LLOC metric change of 0.5 from version 1.0 to version 2.0". The transformed information is stored in a different CSV file for every project. Each file has the name structure "productName" + "-versionGraphData.csv". To generate the metric evolution graphs across several versions, the user has to run the method *createVersionGraphs();* in the same class. If any additional metrics should be added to the version graphs, they need to be added to the *getProductMetricData();* method as keys and to the enumeration *MetricKeysImpl* in the *VersionMetrics* class.

# Summary

In this chapter I would like to present a summarized version of the findings of this thesis and the most important excerpts from the metric correlation analysis. The bachelor thesis “On Correlations between Vulnerabilities, Quality-, and Design-Metrics” is centered around using metric data, derived from static code analysis, to make predictions about the qualities of software code prior to its release. This has many benefits, among others the possibility to save financial and temporal resources that would be required if a software project in the production stage was found to be unstable or vulnerable. To this end, the study focuses on calculating metric data based on the software’s quality, security and vulnerability. Further, it analyzes the effects these different software aspects exert onto each other.

This study is based on a prior study, conducted by Brigitte Wiebe. She created a tool that could gather metric data about a project’s quality and security and calculate their correlation values using the Spearman correlation method. Based on this preexisting foundation, the present thesis defines and answers four research questions:

- RQ1: How can the extraction and evaluation of real-world vulnerability data about software projects be automated?
- RQ2: Can the findings of the previous study be confirmed and how do the vulnerability data metrics correlate with the quality and security metrics of the software project?
- RQ3: How can the process of selecting and extracting the software projects to evaluate be automated?
- RQ4: How does the correlation between the quality, security and vulnerability metrics behave across multiple software versions?

In the following sections I will proceed with short summaries for each of the research questions and their outcomes.

## 9.1 Summary of research question 1

The first research question is about how the vulnerability data of software projects can be automatically extracted, gathered and evaluated. The tool for this study gathers this data from one of the largest vulnerability resources on the internet, the National Vulnerability Database (NVD). In the NVD, software vulnerabilities, submitted by many independent people around the world, are gathered, checked and persisted. They are completed with a lot of metadata, as well as some metric data, calculated by the NVD employees. The database offers downloadable feeds for this information, which the tool makes use of. This is how the tool actually extracts the data that is later needed

for the analysis. After that, it saves this data in a local database, using the product Elasticsearch. Elasticsearch is a useful NoSQL database complete with fuzzy searching functionality as defined in section 8.1. Based on some of the extracted metadata, vulnerability metrics are then defined. These are :

- The number of vulnerabilities for a project
- The number of vulnerabilities per 1000 lines of code in a project
- The average CVSS2 and CVSS3 score for a project
- The maximum CVSS2 and CVSS3 score for a project

The CVSS score is the metric calculated by the NVD upon receiving a vulnerability. It is very representative of its severity and environment and therefore very useful for the correlation analysis. It is described in subsection 2.3.5. The tool has defined methods that return all these vulnerability metric results, given a project name and a corresponding vendor name. The tool can also search based on the project version, which is required for RQ4. This is how the vulnerability data is evaluated.

There are some issues that arise when trying to search for a product and vendor name in the local database. These arise from naming differences between the GitHub pages of the projects, where one would normally gather this data from, and the NVD product name and vendor name fields. These seldom overlap, causing a very low number of matches when searching. Based on a study of the matching process conducted by hand, these are still found with a recall of 50% and a precision of 71%.

## 9.2 Summary of research question 2

The second research question is about reiterating the results from the previous study and studying further correlations based on the vulnerability metrics. The previous study analyzed the internal correlations of the software quality metrics and the correlations between the quality and security metrics. For this question, a very different set of projects was input in the analysis tool. In the summary of RQ3 I explain some of the filters for the choice of these projects and how they were selected automatically, as opposed to statically. Additionally, a different quality metric was added to the analysis, and one security metric was removed.

With the different set of input projects (a total of 33), the correlations were analyzed once again. The results were very similar to those of the previous study, which shows their validity. Namely, the quality metrics showed many significant internal correlations between each other. They clearly deliver a very well-rounded picture of software quality when used together and can be used to illustrate how causing one of the metrics to change can have a strong influence on the other metrics. The main point shown by this is that developers should be taught about these effects and make use of static code

analysis to avoid costly code maintenance. They can be quite sure, for example, that having larger classes on average (as calculated by the LOCpC metric) can cause their code to become very complex (as calculated by the WMC metric).

The results for the correlations between the quality and security metrics were also very similar. There were almost no correlations between them, with negative correlations appearing between some of the metrics. This is appointed to the fact that the security metrics, used in both studies, are very limited to the visibility aspect of security and are not fit to give a general representation of the software security. It was also found, that writing bad quality code, and therefore increasing the code quality metrics, naturally leads to more visibility of method and types being required, which in turn decreases the security metrics. These become necessary, meaning that they do not quality as too generously accessible. The vulnerability metrics, as defined in RQ1, would provide as a great security metric, featuring much more information about their environment and severity. Unfortunately, they cannot be calculated before project releases.

The results for the vulnerability correlations were inconclusive, due to some issues which are beyond the control of this study. Among these are the naming differences for projects between the NVD and GitHub, as mentioned in the first research question summary. Others stem from the fact that many Gradle + Java projects are free android applications, which most people would not necessarily enter vulnerability information for. Due to these and other circumstances, described in subsection 4.4.2, the vulnerability correlations could not be analyzed. The possibility for retrieving them is however demonstrated in chapter 5 as the vulnerability correlation calculation and metric gathering functionalities are all completely implemented into the tool.

## 9.3 Summary of research question 3

The third research question is about selecting and extracting software projects for the metric correlation analysis automatically. To be able to find useful projects that can be analyzed by the metric correlation analysis tool, some prerequisites were defined on a technical level. These are described in detail in section 6.1 and are summarized as follows:

- The projects must be open-source – this is because only free projects should be analyzed for student thesis purposes
- The projects must be in the Java language – this is a requirement based on some technical details of the metric correlation analysis tool
- The projects must use the Gradle build nature – this is a requirement based on some technical features of the metric correlation analysis tool
- The projects must be sufficiently popular – this requirement should increase the validity of the study and find more projects with vulnerability data

- The projects must have retrievable version information – this is a requirement based on RQ2 and RQ4 where the metric correlations should be analyzed specifically between metrics, calculated on the same project versions
- The projects should have some vulnerability data – this point ends up being optional for RQ2, as making it mandatory would result in too few projects

The metric correlation analysis tool makes use of the GitHub API to enforce these prerequisites in the selection process and therefore automates the project selection input. It allows for a larger number of projects to be analyzed and is easily extendable, should new prerequisites be defined or some removed. Instructions for how to do this are found in chapter 8. With these prerequisites, without the optional one, 3,200 projects qualify for the metric correlation analysis. Including the optional one, about 10 projects can be found. The total number of considered projects for this study was 9,500. Some ideas on how to further increase the number of analyzed projects can be found in chapter 10.

## 9.4 Summary of research question 4

The fourth research question is about analyzing the trends in the metric evolution of the quality, security and vulnerability metrics. This question was also considered in the previous study (except the vulnerability metrics), but was only described shortly with one project in mind. For the present study, six projects with several differing numbers of versions were analyzed. The functionality of the tool to automatically graphically represent the project metric evolution was also implemented. This means that in the future, any number of projects can undergo the metric evolution trend analysis without changing the metric correlation analysis tool.

The trends of the quality metrics and their internal correlations largely overlapped with the correlation values calculated in RQ2. This is another outcome that confirms the findings of both studies to be coherent. The trends of the security metrics, compared to the quality metrics, was also largely similar to what was to be expected, based on the results of the second research question. Some propositions of why the trend evolution took this course were also made across all six projects. Again, the trends of the vulnerability metrics are not enough to make assumptions about their correlations with the quality and security metrics, but they have demonstrated the ability of the tool to deal with this input and represent it as well.

# Future work

There are several ways to continue the work of this thesis and extend the metric correlation analysis tool. Some of them would result in a larger number of projects studied and some would result in even more information about their metric correlations.

## 10.1 Analyzing more projects

The metric correlation analysis tool already implements an automatic way of searching for the studied projects, given some prerequisites. More projects could be analyzed if:

- More computational time is devoted to the project search and less popular projects are allowed. Currently the top 8900 most popular java projects on GitHub are studied to filter the ones that fit all the study prerequisites. If this number is increased, even more projects could be studied for metric correlations. One thing to be wary of is the assessment that allowing more projects to be studied would also involve less popular projects. Projects with a lower popularity are more likely to be test or student projects made for learning purposes. Scanning a couple of thousand more projects would probably not harm the overall popularity. This could be tested with the *getAverageNumberOfStars();* method of the *ProjectSelector* class. At the time of writing this study, the number of stars for the discovered projects average at 3911.

- The metric tools and definitions could be exchanged so that languages, other than Java are allowed. Possibly, there are metric tools that work on other object-oriented languages, like Python or C++. Integrating them into the tool would require substantial code changes, but it is not altogether impossible. Allowing for projects in other languages to be studied would further increase the sample size.

- Allowing Maven projects to be studied. Maven is a software project management and comprehension tool, which can navigate the project's build process much in the same way that Gradle does. Currently, the tool builds the projects with Gradle and runs the metric analysis tools on them. Some of them require this explicitly. If other tools are found, that do not require this, or other metrics are calculated altogether, then this prerequisite can be dropped. Studying both Maven and Gradle projects could potentially double the project sample size. At the time of writing this thesis Gradle is seven times more popular than Maven (based on GitHub star rating), but Maven users still represent a very large group in this market.

All changes that are related to applying the project selection prerequisites, can be implemented in the *ProjectSelector* class. Of course, any increase in the project sample size will give more statistical significance to the findings of the future studies.

Another way to increase the amount of studied projects would be to find more projects with version information. Projects that fit all the study prerequisites are still impossible to study properly without sufficient version information. The tool currently utilizes the GitHub "releases" feature to find the version information of the projects. This information consists of two things – the version number and the repository commit that relates to this specific number. Not all the projects that the tool initially discovers have their release information input into GitHub in this way, as using this functionality is completely optional for the developers. However, over 70% of the projects, chosen by the tool, do utilize it in fact. Therefore this method was still a fitting choice for this particular study. A future work suggestion, which would include projects that do not use this functionality, would be the following: the version information could be retrieved from the meta files included in the projects. For example, the "build.gradle" file, located in Gradle-based projects, can hold the version information as well. An idea would be to scan each commit of a project sequentially, until the version number in the build file changes. A change would mean that a new version has begun and its respective commit id can be identified. Any such improvements can be added to the *ProjectsOutputCreator* class, which creates a CSV file containing all the project and version data for the metric correlation analysis.

## 10.2 Studying more correlations

There is definitely room left for more security metrics to be added to the correlation analysis. While it does seem that the quality metrics deliver a coherent and even correlating result when paired with the vulnerability metrics, the security aspect does not achieve that. The IGAM and IGAT metrics are very starkly focused on the visibility aspect of security. It will be interesting, and it may reveal some further positive correlations, if more security metrics would be added. The problem with finding security metrics is that they are often viewed as synonymous to the quality metrics, but this should not be the case. There are other ways of assessing security, rather than quality being researched [Wan+11].

## 10.3 Predicting vulnerabilities

Ultimately the findings of this thesis are meant to contribute the efforts of predicting software vulnerabilities, before a new version of said software releases. Knowing the benefits of static code analysis in determining the odds of security gap openings would be of crucial benefit to developers. They could use this to prevent losing time and money, the most valuable resources they, and the companies they work for, have. The results of this thesis can be used to deepen any research in this direction. The metrics of the study can be changed to find the most defining quality, security or even other metrics that correlate with the presence of vulnerabilities in the software systems.

## Bibliography

[Abe+06] Muhammad Abedin et al. “Vulnerability analysis For evaluating quality of protection of security policies”. In: *Proceedings of the 2nd ACM workshop on Quality of protection - QoP ’06* (2006), p. 49. DOI: 10.1145/1179494.1179505. URL: http://portal.acm.org/citation.cfm?doid=1179494.1179505.

[Ame89] Pierre America. “Object-oriented software construction”. In: *Science of Computer Programming* 12.1 (1989), pp. 88–90. ISSN: 01676423. DOI: 10.1016/0167-6423(89)90034-8. URL: http://linkinghub.elsevier.com/retrieve/pii/0167642389900348.

[BBL76] B W Boehm, J R Brown, and M Lipow. “Quantitative Evaluation of Software Quality”. In: *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE ’76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 592–605. URL: http://dl.acm.org/citation.cfm?id=800253.807736.

[BG94] Michael Buckland and Fredric Gey. “The Relationship between Recall and Precision”. In: 45.1 (1994), pp. 12–19.

[BRI16] JUXHIN DYRMISHI BRIGJAJ. *No Title*. 2016. URL: https://www.acunetix.com/blog/articles/whats-new-in-cvss-version-3/.

[Bro+98] William J Brown et al. “AntiPatterns: Refactoring Software , Architectures, and Projects in Crisis”. In: *Crisis* 3.4 (1998), pp. 281–284. ISSN: 0849329949. URL: http://www.amazon.com/dp/0849329949.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. “A Metrics Suite for Object Oriented Design”. In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. ISSN: 00985589. DOI: 10.1109/32.295895.

[Coh92] Jacob Cohen. “Statistical power analysis”. In: *Journal of Clinical Psychiatry* 1.3 (1992), pp. 98–101.

[Cow15] David Cowan. “The Affordable Ten-Step Plan for Survival in Cyberspace”. In: (2015).

[CVE] CVE. *CVE Website*. URL: https://cve.mitre.org/.

[dev18] developer.android.com. *developer.android.com*. 2018. URL: July%2027,%202018.%20https://developer.android.com/.

[Fir] First.org. *No Title*. URL: https://www.first.org/cvss/specification-document.

[Fow02] Martin Fowler. “Refactoring: Improving the Design of Existing Code”. In: (2002). ISSN: 14359448. DOI: 10.1007/s10071-009-0219-y. arXiv: 9809069v1 [gr-qc].

[Fre+06] Stefan Frei et al. "Large-scale vulnerability analysis". In: *SIGCOMM workshop on Large-scale attack defense (LSAD)* (2006), pp. 131–138. DOI: 10.1145/1162666.1162671. URL: http://dl.acm.org/citation.cfm?id=1162671.

[Gha+13] Hamza Ghani et al. "Predictive vulnerability scoring in the context of insufficient information availability". In: *2013 International Conference on Risks and Security of Internet and Systems, CRiSIS 2013* (2013). DOI: 10.1109/CRiSIS.2013.6766359.

[Git] GitHub. *GitHub*. URL: www.github.com.

[Goo] Google. *Android developer manual*. URL: https://developer.android.com/studio/build/.

[Hat] Laura Hatula. "Programmers are copying security flaws into your software, researchers warn". In: *CNET* ().

[Hov+12] Aram Hovsepyan et al. "Software vulnerability prediction using text analysis techniques". In: *Proceedings of the 4th international workshop on Security measurements and metrics - MetriSec '12* (2012), p. 7. DOI: 10.1145/2372225.2372230. URL: http://dl.acm.org/citation.cfm?doid=2372225.2372230.

[JSO] JSON. *JSON org*. URL: https://www.json.org/index.html.

[Kan02] Stephen H Kan. *Metrics and Models in Software Quality Engineering*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201729156.

[KRK17] D. Richard Kuhn, M. S. Raunak, and Raghu Kacker. "An Analysis of Vulnerability Trends, 2008-2016". In: *Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C 2017* (2017), pp. 587–588. DOI: 10.1109/QRS-C.2017.106.

[Mcc76] Thomas J Mccabe. "A Complexi-ty". In: 4 (1976), pp. 308–320.

[McG04] G. McGraw. "Software security". In: *IEEE Security Privacy* 2.2 (2004), pp. 80–83. URL: https://ieeexplore.ieee.org/abstract/document/1281254.

[MSA08] Misra, Sanjay, and Akman. "Weighted Class Complexity: A Measure of Complexity for Object Oriented System". In: *JOURNAL OF INFORMATION SCIENCE AND ENGINEERING* 24 (2008), pp. 1689–1708.

[MSR06] Peter Mell, Karen Scarfone, and Sasha Romanosky. "Common Vulnerability Scoring System". In: *IEEE Security and Privacy Magazine* 4.6 (2006), pp. 85–89. ISSN: 1540-7993. DOI: 10.1109/MSP.2006.145. URL: http://ieeexplore.ieee.org/document/4042667/.

[NPP13] Ameya Nayak, Anil Poriya, and Dikshay Poojary. "Type of NOSQL Databases and its Comparison with Relational Databases". In: *International Journal of Applied Information Systems* 5.4 (2013), pp. 16–19. DOI: 10.5120/ijais15-451326.

[NVD] NVD. *NVD Website*. URL: www.nvd.nist.gov.

[OE07] Andy Ozment and D Software Engineering. "Improving Vulnerability Discovery Models Problems with Definitions and Assumptions Categories and Subject Descriptors". In: *Cycle* (2007), pp. 6–11. ISSN: 15437221. DOI: 10.1145/1314257.1314261.

[Pel+16] Sven Peldszus et al. "Continuous Detection of Design Flaws in Evolving Object-oriented Programs Using Incremental Multi-pattern Matching". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. New York, NY, USA: ACM, 2016, pp. 578–589. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970338. URL: http://doi.acm.org/10.1145/2970276.2970338.

[Pro] Open Web Application Security Project. *OWASP dependency check*. URL: https://www.owasp.org/index.php/OWASP_Dependency_Check.

[RZ13] Sanaz Rahimi and Mehdi Zargham. "Vulnerability scrying method for software vulnerability discovery prediction without a vulnerability database". In: *IEEE Transactions on Reliability* 62.2 (2013), pp. 395–407. ISSN: 00189529. DOI: 10.1109/TR.2013.2257052.

[Sch+00] M Schumacher et al. "Data Mining in Vulnerability Databases". In: *Network* (2000).

[Shi+11] Yonghee Shin et al. "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities". In: *IEEE Transactions on Software Engineering* 37.6 (2011), pp. 772–787. ISSN: 00985589. DOI: 10.1109/TSE.2010.81.

[SO] Doina Caragea Su Zhang and Xinming Ou. *An Empirical Study on Using the National Vulnerability Database to Predict Software Vulnerabilities*, pp. 217–232. ISBN: 9783642230882.

[Von16] Sarah Vonnegut. *Security Testing in the SDLC: A Beginner's Guide*. 2016. URL: https://www.checkmarx.com/2016/02/26/security-testing-sdlc-beginners-guide/.

[Wan+09] Ju An Wang et al. "Security metrics for software systems". In: *Proceedings of the 47th Annual Southeast Regional Conference on - ACM-SE 47* (2009), p. 1. DOI: 10.1145/1566445.1566509. URL: http://portal.acm.org/citation.cfm?doid=1566445.1566509.

[Wan+11] Ruyi Wang et al. “An improved CVSS-based vulnerability scoring mechanism”. In: *Proceedings - 3rd International Conference on Multimedia Information Networking and Security, MINES 2011* (2011), pp. 352–355. DOI: 10.1109/MINES.2011.27.

[Wav15] Forrester Wave. “Forrester Wave: Application security report”. In: (2015).

[Wie17] Brigitte Wiebe. “Bachelorarbeit Eine empirische Studie uber die Korrelation zwischen Sicherheitsschwachstellen und Qualitätseigenschaften von Software-Designs”. In: (2017).

[Wil98] Hays W. SSkip William H. Brown Raphael C. Malveau. “AntiPatterns: Refactoring Software, Archi- tectures, and Projects in Crisis.” In: (1998).

[WSS14] James Walden, Jeff Stuckman, and Riccardo Scandariato. “Predicting vulnerable components: Software metrics vs text mining”. In: *Proceedings - International Symposium on Software Reliability Engineering, ISSRE* (2014), pp. 23–33. ISSN: 10719458. DOI: 10.1109/ISSRE.2014.32.

[Y06] Dodge Y. *The Oxford Dictionary of Statistical Terms*. 2006. ISBN: ISBN 0-19-920613-9.

[YB07] Li Yujian and Liu Bo. “A normalized Levenshtein distance metric”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29.6 (2007), pp. 1091–1095. ISSN: 01628828. DOI: 10.1109/TPAMI.2007.1078.

[ZNW10] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. “Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista”. In: *ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation* (2010), pp. 421–428. ISSN: 2159-4848. DOI: 10.1109/ICST.2010.32.

[ZS12] C Zoller and A Schmolitzky. “Measuring Inappropriate Generosity with Access Modifiers in Java Systems”. In: *Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement(IWSM-MENSURA)*. Vol. 00. 2012, pp. 43–52. DOI: 10.1109/IWSM-MENSURA.2012.15. URL: doi.ieeecomputersociety.org/10.1109/IWSM-MENSURA.2012.15.