Universiteit Leiden

ICT in Business

Towards a Metric-based Security Model

Name: Konstantinos Vlyssidis
Student-no: s1418572

Date: 23/01/2016

1st supervisor: Dr. W. Heijstek
2nd supervisor: Prof. Dr. A. Plaat

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

This study is an initial attempt to create a methodology that assesses a system's software security by means of automatically-obtained vulnerability metrics. The motivation behind this study lies on the fact that although several security metrics already exist, relevant research is still on an early stage, mainly due to the difficulty in defining meaningful and efficient metrics. Furthermore, no single quality metric or measurement is able to holistically assess the security status of a system. Due to this reason, a framework of metrics is needed to combine multiple security measurements.

Using automated measurements provided by static analysis tools, three metrics were created to assess the susceptibility of the software system on different types of software vulnerabilities both on its source code as well as on its third party dependencies. Furthermore, a fourth metric was integrated which assesses the update behavior of the external dependencies of the system.

The first two metrics, "Acute Issues" and "Latent Issues", detect vulnerabilities within the source code of the system and provide a rating based on the number of vulnerabilities detected. The other two metrics detect issues located in third-party dependencies of the system. More specifically, the "Dependency Vulnerability" metric, as the name states, detects vulnerabilities located on a system's dependencies and returns a rating based on the number of vulnerabilities found and their severity. Finally, the "Dependency Freshness" metric, detects how recently the third-party dependencies of the system were updated.

The evaluation of the framework was conducted in two parts. In the first part, four software security consultants and developers were interviewed to check the model against significant user acceptance determinants for methodologies, such as usefulness and compatibility, as well as to verify the metrics' conformance to validity criteria such as discriminative power, predictability and consistency. In the second part of the validation, a longitudinal analysis was conducted in order to verify the tracking ability of the source code metrics.

The results show that the framework is generally considered useful among the interviewees, with an average of 4.5 out of 5 in a likert scale. Furthermore, the proposed methodology is compatible to the way the interviewees currently assess the security of software systems (3.75 out of 5). Moreover,

the data gathered suggests that the metrics comply to the validity criteria as defined by ISO 1061-1998.

Acknowledgements

The completion of this thesis brings a closure on a physically and mentally intense, and yet mesmerizing period. During these two and a half years, in which I followed the "ICT in Business" MSc program in the University of Leiden, there are quite a few people I would like to thank and acknowledge as contributing factors towards my progress.

Starting, I would like to express my gratitude to both my university supervisors, Dr. W. Heijstek and Prof. Dr. A. Plaat for guiding me in the making of this thesis project. Furthermore, I would like to praise my company supervisors Dr. H. Xu, Dr. B. Vieira and Prof. J. Visser, as well as all SIG employees in general, for trusting me to research this topic and providing me with valuable advice and tutoring.

This period though has been much more than just an academic experience. Hence, I would like to thank Antigoni and friends such as Stefanos, Christos, Bill, Cristian and Katia for being by my side and sharing wonderfull moments.

First and foremost, however, I am greatly indebted to my family for shrouding me with full support and trust.

Contents

Chapter 1

Introduction

System and software security has been a significant area of attention in the field of computer science. Its importance is steadily increasing as software becomes a vital part on most aspects of everyday operations. Security-related incidents, however, do not seem to be decreasing in number or on impact.

If anything, recent reports indicate the contrary. The number and size of security breaches has been on a steady rise in the previous years [65][68][66][69]. The incidents spread across industries irrelevant of their type and affect organizations of all sizes [65],[66]. Additionally, the attacks can be initiated both by internal and external actors for ideological, financial, espionage or even entertainment reasons [65]. More worryingly, what is noted on the reports is that the time needed to compromise an asset is significantly smaller than the time needed to discover the breach; and this difference is increasing [65].

An effective and continuous assessment of the security status of an application during its lifecycle, in order to identify and reduce risk as soon as possible, is vital both on correcting weaknesses before production as well as on reducing the time to discovery of a weakness.

Eliminating vulnerabilities found in the source code or external dependencies of a system is an essential way of reducing such a risk [66]. CVSS [49] defines a vulnerability as a bug, flaw, weakness, or exposure of an application, system, device, or service that could lead to a failure of confidentiality, integrity, or availability. Vulnerabilities can be introduced during the development or maintenance of the software, due to human errors, ineffective design of the specifications of the system, non-adherence to best practices, etc. Additionaly, they can vary significantly, ranging from an incorrect or

missing user input validation to errors in security features such as insecure randomness. A number of attempts has been made, for that reason, to organize vulnerabilities based on their type or severity such as the "Seven Pernicious Kindgoms" taxonomy [73], the "19 Deadly Sins of Software Security" [35] and the "OWASP Top 10" list [57].

Testing, executing a program at a prescribed way to see whether it functions properly, is vital in developing systems. However the former, by itself, is often ineffective in detecting security errors. Attackers attempt to exploit weaknesses that system designers did not consider, and standard testing is unlikely to uncover such weaknesses [28].

A number of additional approaches to detect vulnerabilities exist. Code reviews, the manual inspection of code for defects and improvement opportunities, is the most basic form of a vulnerability detection procedure. Code review methods vary in formality, rigor, effectiveness, and cost [81], and provide benefits on the system's quality, on the project's time to market and overall cost, as well as on the spread of product, project and technical knowledge among the team members[81].

Manually inspecting a system, however, is a time-intensive process [81]. Furthermore, successfully and consistently identifying errors depends on the reviewer's skills. Automated tools exist, therefore, to minimize the human time and complexity needed during code analysis [16]. These tools vary regarding the way they assess the system (static and dynamic analysis tools), the areas of the system they assess and the techniques they use, offering tradeoffs between the required effort and analysis complexity [28]. Research on the tools has indicated that their use can lead to significant quality improvement and cost savings [16], as well as reduce the programming experience required to detect security vulnerabilities [17].

The aforementioned tools, though, generate a list of potential vulnerabilities. An enumeration of technical findings, however, often fails to accurately describe and quantify the security status of a system. Such a quantification is useful in order to provide a basis to assess the quality of coding within the application, set objectives and thresholds that the application should align with, determine information like the amount of work needed to improve the system, the areas in which to focus, etc.

The concept of the software quality metric becomes relevant in this case. A software quality metric is defined as a function whose input is software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality [13]. Metrics are used to facilitate decision making and improve

performance and accountability through collection, analysis and reporting of relevant performance-related data [67]. Although several security metrics already exist ([21],[45],[27],[80]), research is still on an initial stage [56][9], mainly because of the difficulty in defining meaningful and efficient metrics. Additionally, as Manadhata et al. [45] notes, no single quality metric or measurement is able to holistically assess the security status of a system. Due to this reason, the author states, a framework of metrics is needed to combine multiple security measurements.

1.1 Research Questions

The main research question is formulated as below:

How can software security be expressed by means of automatically-obtained vulnerability metrics?

In order to answer the main research question the following sub-questions have been determined:

1. *How to derive metrics from a system's software vulnerabilities?*
   (a) *Can we assess a system's security by deriving metrics that rely on the severity of software vulnerabilities?*
   (b) *Can we derive metrics from static analysis tools' automated measurements?*
2. *How to build a framework of metrics to assess software security?*
3. *Is the proposed framework useful for assessing software security?*
4. *Do the proposed metrics accurately express software security?*

Our main goal is to quantitatively assess a system's software security. In order to do so, we aim at creating a framework which will consist of metrics derived from automated measurements. These metrics will provide a rating based on software vulnerabilities existing in the system.

In order to define the metrics we will have to classify software vulnerabilities based on some common characteristics of the latter, such as their severity and their location within the system. Furthermore, we wish to explore whether we can use static analysis tools to detect vulnerabilities and derive metrics from the tools' output.

Finally, we intend to validate the proposed framework and its metrics to determine whether the latter measure correctly the security conditions they should measure and whether the framework can assist in assessing a software system's security level.

1.2 Thesis Structure

The thesis is organized as following. Chapter 2 presents relevant to our research background, regarding software security, software vulnerabilities and static analysis tools, as well as related work regarding software security metrics and frameworks.

Chapter 3 presents the methodology followed during our research, i.e. the steps followed to derive the framework of metrics. Chapter 4 introduces the design of the framework, concerning topics such as its architecture, generic principles used in the rating method of its metrics and the dataset of systems that was used during the framework's development and testing.

Chapters 5 and 6 present specific information regarding each metric used. Chapter 5 focuses on the source code metrics. It provides the tools that were used, the steps followed to separate the vulnerability patterns between each metric, as well as their rating methods. Furthermore, it provides the result of the application of the metrics on the dataset of systems used.

Chapter 6 focuses on the external dependencies metrics. It starts by presenting the "Dependency Vulnerability" metric, the tools that it uses as well as its rating method. Afterwards, it provides information regarding the "Dependency Freshness" metric and finally it gives the results of the application of the metrics in the dataset of systems, as well as the results of the analysis conducted to identify any correlation between the ratings of the four metrics and maintainability characteristics of the systems.

In Chapter 7 the validation of our approach is presented. The chapter starts by introducing the design of the validation procedure of the framework, followed by a presentation of the latter's results. Chapter 8 provides a discussion of the findings of this study, as well as the threats to its validity. Finally the conclusion of the thesis is drawn in Chapter 9.

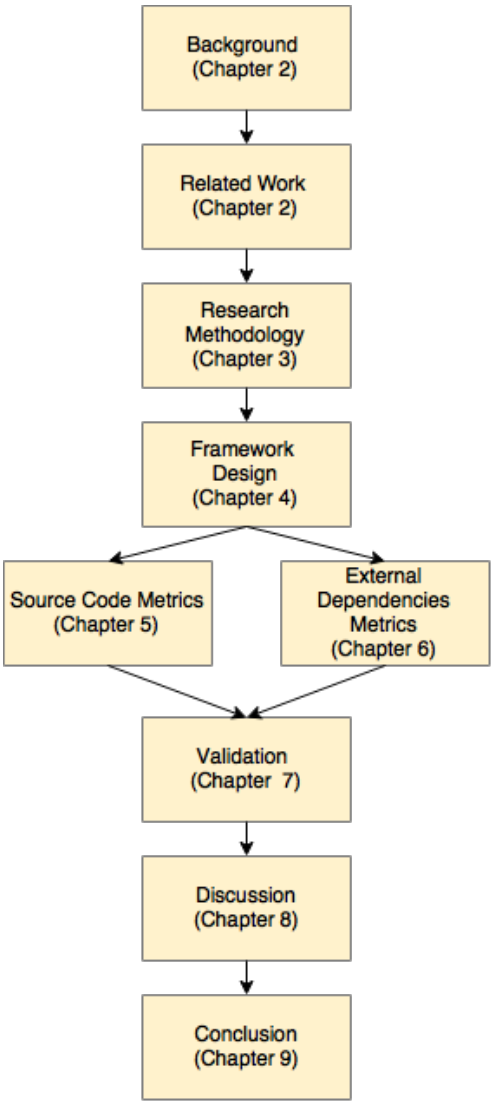


Figure 1.1: Thesis Structure

Chapter 2

Background

In this chapter, concepts, aspects and research, relevant to this thesis, will be presented. At the beginning it is considered beneficial to define software security and describe fitting perspectives such as software vulnerabilities and vulnerability detection tools. Afterwards, an elaboration is provided on what a software quality metric is and, finally, related work is discussed, on software security metrics and frameworks.

2.1 Software Security

2.1.1 Definition

Software security, as any quality attribute, is an intangible concept. A clear definition of security, and thus, of important requirements that software should align with, in order to be considered secure, is vital to successfully assess and measure the security level of an application.

The ISO 25010-2011 [38] model specifies eight product quality characteristics. Security, one of the characteristics, is defined as the degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization. Five subcharacteristics of security are proposed: confidentiality, integrity, non-repudiation, accountability and authenticity.

The ISO standard does not include in the definition of security the availability of the system, which is included, indeed, in the reliability charac-

teristic. Availability is a principle of the classic CIA triad (standing for Confidentiality, Integrity, Availability), one of the simplest but also most widely applied security models. Other approaches also have been made to define software and information security like the ones contained in the CNSS National Information Assurance Glossary [25] and the Parkerian Hexad [58].

2.1.2 Vulnerabilities

One of the main ways of affecting the security of a software system is by exploiting vulnerabilities found in its source code or related dependencies [65]. CVSS [49] defines a vulnerability as a bug, flaw, weakness, or exposure of an application, system, device, or service that could lead to a failure of confidentiality, integrity, or availability. Pfleeger [23] uses a more vague definition of a vulnerability as a weakness in the security of a computer system that might be exploited to cause loss or harm. Landwehr et. al [43], finally, define a computer security flaw as any condition or circumstance that can result in denial of service, unauthorized destruction of data, or unauthorized modification of data.

As explained in [53], security vulnerabilities have two distinct features when compared with other categories of bugs: they can severely affect an organization's infrastructure [63] and they can cause significant financial damage to an organization [16] [72]. Additionally, a further consequence of a vulnerability disclosure is a negative and notable change in market value for a software vendor [71] and a harm in reputation.

There are various types of vulnerabilities, differing on how they are introduced in the system (such as vulnerabilities originating from improper user input validation and vulnerabilities caused by weak encryption mechanisms), their impact once they are exploited (which can range from a mere stylistic annoyance in the appearance of a website, to theft of personal and financial information), or the complexity needed and probability to be exploited. CVE identifiers [51] are an effort to create unique, common identifiers for publicly known cyber security vulnerabilities. Those identifiers are then integrated in a list of information security vulnerabilities and exposures with the goal of making the sharing of data across separate vulnerabilities capabilities easier. Additionally, extensive research has been performed in creating a taxonomy of vulnerabilities.

The Common Weakness Enumeration (CWE) [52] is such a taxonomy of common software weaknesses that can occur in the architecture, design or development of a software and can lead to exploitable security vulnerabilities. The CWE list can be used as a standard for evaluating software security

tools and as a common baseline for weaknesses identification, mitigation and prevention.

In the "Seven Pernicious Kindgoms" taxonomy [73], the authors motivate their work by stating that such a classification of vulnerabilities can assist developers in easier recognition of categories of problems and identification of existing errors during the development of software. Subsequently, they split vulnerabilities in eight groups, seven of which are dedicated to errors in the source code and one is related to configuration and environment issues. These, in order of importance, are:

1. **Input Validation and Representation**: Problems resulting from trusting input and caused by metacharacters, alternate encodings, numeric representations, etc. Issues such as SQL Injection, Cross-Site scripting and Command Injection belong in this category.
2. **API Abuse**: Issues resulting from improper communication with APIs, caused by dangerous API functions, false assumptions regarding the behaviour of APIs, etc.
3. **Security Features**: This category includes topics such as authentication, access control, confidentiality, cryptography, and privilege management.
4. **Time and State**: Defects related to unexpected interactions between threads, processes, time, and information. These interactions happen through shared state.
5. **Errors**: Errors related to error handling.
6. **Code Quality**: Poor code quality leads to unpredictable behavior.
7. **Encapsulation**: Issues originating from insecure boundaries between components or data.
8. **Environment**: Issues that are outside of the source code but are still critical to the security of the product.

The OWASP Top-10 [57], is another classification, listing the 10 most dangerous types of vulnerabilities, as defined by OWASP, and ranked based on their exploitability, prevalence, detectability and impact. The entries on the OWASP Top-10 list are:

1. **Injection**: Injection flaws, such as SQL, OS, and LDAP injection.

2. **Broken Authentication and Session Management**: Not correctly implemented application functions related to authentication and session management.

3. **Cross-Site Scripting**: XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping.

4. **Insecure Direct Object References**: Lack of an access control check or other protection, which enables attackers to manipulate direct object references to access unauthorized data.

5. **Security Misconfiguration**: Insecure configuration defined and deployed for the application, frameworks, application server, web server, database server, or platform.

6. **Sensitive Data Exposure**: Ineffective protection of sensitive data, such as credit cards, tax IDs, and authentication credentials.

7. **Missing Function Level Access Control**: Applications need to perform function level access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization.

8. **Cross-Site Request Forgery**: A CSRF attack forces a logged-on victims browser to send a forged HTTP request, including the victims session cookie and any other automatically included authentication information, to a vulnerable web application.

9. **Using Components with Known Vulnerabilities**: Applications using components, such as libraries, frameworks, and other software modules, with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.

10. **Unvalidated Redirects and Forwards**: Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

Other taxonomies consist of works like the "19 Deadly Sins of Software Security" [36], the extended "24 Deadly Sins of Software security" [37], the taxonomy of Unix vulnerabilities presented by Bishop [19], etc.

2.1.3 Static Analysis vulnerability detection tools

Automatic static analysis tools have been an area of research for many years and recently added to security engineering processes, such as the Security Development Lifecycle [44] .

Any tool that analyzes source code without executing it is performing static analysis [24]. These tools are used to detect the most common errors in a particular coding language and assist the developer to create more stable, reliable and secure code. There exist multiple types of static analysis tools, offering different functionality, required effort and analysis complexity, ranging from simple type checkers to full program verifiers that attempt to prove complex properties about programs.

Extensive literature exists on verifying the accuracy, effectiveness and usefulness of static analysis tools. In general, the researchers agree that the former can detect faults (e.g [28],[16],[64],[15]) with varying conclusions though, as to the degree of the effectiveness (e.g. the false positive rate of 30-100% found in [42] in comparison to the false positive rate of 5-22% found in [16]) and significance of the errors detected. Existing research, additionally, is useful when trying to identify the advantages and disadvantages of static analysis tools.

By automating source code inspections the human time and complexity needed can be reduced [16]. Furthermore, since static analysis tools do not need to execute the application, an analysis can be completed at an earlier stage in the lifecycle of the system. The decrease of the overall cost of a project is one of the benefits of such a move, as Baca et al. [16] demonstrate by conducting a case study in which mature software with known vulnerabilities was subjected to a static analysis tool. In his research the author found that an average of 17% cost savings would have been possible by the use of the static analysis tool. Moreover, earlier error detection can result in less human resources and time needed to detect the problem, correct it and apply the correction, due also to the reason that static analysis detects the root causes of vulnerabilities rather than the symptoms [47]. Another advantage, as noted in the same article, is that the introduction of new vulnerabilities, caused by post-production patching, can be avoided.

The use of static analysis tools can additionally reduce the development experience required to identify programming errors. Baca et al.[17] drew this inference by conducting an experiment in which a combination of security and static analysis tools experience was found to be the most effective in detecting security vulnerabilities. Another benefit noted by Evans et al. [28] is the capability of these tools to eliminate errors, during the code

review phase, caused by human oversights. Finally, as elaborated in the same article, an important advantage of static analysis, compared to test-case execution, is the ability to validate all possible program executions (resulting from direct source code analysis), a significant improvement from a security perspective.

However, the usage of static analysis tools entails certain disadvantages as well. These tools are not able to detect conceptual errors [28] as well as errors on the design and architecture of the system. The dubious efficiency of the default issues prioritization decided by the tools' author has also been criticized by researchers [40].

The most important disadvantage, though, of static analysis tools is the false positives rates produced [12]. A warning is considered a false positive if its statement is considered wrong or if the developer does not believe it needs correction [17]. The percentage of false positives varies between different studies. In [79] the authors compared the effectiveness of three bug finding tools with a team review inspection, concluding that bug finding tools can find a subset of the defects found by reviews; however the types can be detected and analyzed more thoroughly. They reported false positive rates higher than 30% for all the tools. In [16] a 20% rate is reported whereas in [42] false positives are claimed to reach 30-100%.

Extensive research has been made, as a result, in an effort to reduce the effect of false positives. In [64] the author suggest an error ranking scheme based on defect likelihood combined with a self-adaptive improvement procedure to readjust the defect likelihood of different bug patterns. In [40] and [41] warning categories are prioritized by analyzing the software change history, whereas in [30] and [31] Heckman et al. have introduced benchmarks that use specific correlation algorithms and classification techniques to evaluate alert prioritization approaches.

2.2 Software Quality metrics

Software quality is hard to specify and measure. ISO standards [38] [13] interpret quality as a desired combination of attributes, such as security, that need to be clearly defined in order to be assessed. Software quality metrics can be identified and used in order to measure the state of these attributes.

ISO 1061-1998 [13] defines a software quality metric as a function whose inputs are software data and whose output is a single numerical value that

can be interpreted as the degree to which software possesses a given attribute that affects its quality. According to George Jelen of the International Systems Security Engineering Association a good metric is Specific, Measurable, Attainable, Repeatable and Time-dependent ("SMART")[39]. Payne [59], additionally, notes that security metrics should indicate the degree to which security goals are being met, in order to be useful.

The purpose of software quality metrics is to make assessments, throughout the software's life-cycle, as to whether the software quality requirements are being met [13]. In this way, subjectivity is reduced, since a quantitative basis for decision-making is provided. Security metrics can be objective or subjective, static or dynamic, absolute or relative, direct or indirect [74]. Another dimension is the level of abstraction of the metrics, which ranges from high-level and risk aware (whose intended audience is management) to low-level development oriented metrics (for developers) [33]. Relevant work exists, as a result, towards a taxonomy of security metrics [59],[61],[74].

The capability though, of security metrics to accurately present security phenomena has, been criticized in some contributions. In designing a security metric, one has to be conscious of the fact that the metric simplifies a complex socio-technical situation down to numbers or partial orders [61]. The side-effects of such a simplification and the lack of scientific proof were also noted by McHugh [48] and McCallam [46]. Burris [20], at last, also notes the role of luck as a source of challenge, especially in the weakest links of information security solutions.

2.2.1 State of the art on software security metrics

In the context of this thesis we consider as relevant to our work, metrics that are low level, quantitative and development-oriented. Metrics belonging in this category are presented below.

Manadhata & Wing [45], propose to use a software system's attack surface measurement as an indicator of the system's security. A system's attack surface is the subset of the system's resources (methods, data, and channels) potentially used in attacks on the system. In this direction, the authors implemented an automated metric to measure the attack surface in a systematic manner. Furthermore, the method was demonstrated by measuring the attack surfaces of small desktop applications and large enterprise systems, implemented in C and Java, and three exploratory empirical studies were conducted as validation. An attack surface metric is a good indicator of the likelihood of an attack to a system. However, such a metric alone, can not provide information regarding the impact of an attack, or possible

vulnerabilities of the system.

Voas et al. [78] propose a relative security metric based on a fault injection technique. Different threat classes of a system were simulated by mutating program variables during the execution of the system and then observing the impact of the threat classes on the behavior of the executing system in terms of successful intrusions. Additionally, a Minimum-Time-To-Intrusion (MTTI) metric based on the predicted period of time before any simulated intrusion can take place was proposed. The implementation of this metric though requires the execution of the system and thus cannot happen in the early stages of its lifecycle. Furthermore, such a metric does not provide information regarding the impact of an attack, or possible vulnerabilities of the system.

Cox et al. [27] analyze the dependency update behavior of industry systems and introduced a benchmark-based security metric to quantify how outdated a system's dependencies are in a whole. Moreover, the author investigated the degree of relationship between a system's dependency update behavior and the dependencies security status. Systems with outdated dependencies were found more than four times as likely to have security issues in their external dependencies. The metric uses a form of static analysis to investigate the update behavior of a system's external dependencies. Furthermore, this metric can be considered as a proactive metric, with a goal to avoid the exploit of vulnerabilities that have already been corrected in later versions of the external dependencies.

Wang et al. [80] attempt to quantify the likelihood of potential multi-step attacks that combine multiple vulnerabilities. By using an attack graph, a model of causal relationships between vulnerabilities, the author proposed an attack graph-based probabilistic metric and studied its efficient computation.

Alhazmi et al. [9] investigated the possibility of predicting the number of vulnerabilities that can potentially be present in a software system but not found yet. In this direction, the author introduced three metrics:

1. **Vulnerability density**: the number of vulnerabilities in the unit size of a code.

2. **Known vulnerability density**: the number of known vulnerabilities in the unit size of a code.

3. **Residual vulnerability density**: defined as the difference between the vulnerability density and the known vulnerability density.

By analyzing data on vulnerabilities discovered in complex software systems the author examines the dynamics of vulnerability discovery rate to see if models can be developed to project future trends. The results indicated that the values of vulnerability densities fell within a range of values and thus it is possible to model the vulnerability discovery by using a logistic model. These metrics can, also, be used from the early stages of the system's lifecycle. However, this implementation does not provide a way to of discovering a system's vulnerabilities, but rather of predicting the existence of future vulnerabilities, based on existing data.

Gilliam et al. [29], finally, also try to provide system admininstrators metrics by creating a vulnerability matrix, a dataset ranking severity of vulnerabilities against frequency of occurrence by platform, to be used by the National Aeronautics and Space Administration. The matrix suggests where to best expend effort in minimizing security risks in the computing environment. The development and maintenance of the metric though is not automated.

2.2.2 State of the art on software Security metrics frameworks

The significance of a framework to integrate and correlate multiple security metrics has been mentioned in a number of articles (e.g [45], [74]). At the same time, though, a lack of such frameworks ([45],[33]) is also apparent. Relevant work, towards this direction, is presented below.

Heyman et al. [33] proposed an approach that leverages the use of security patterns, to improve the development of secure software, and associated security metrics to security patterns. Such a move, the author supports, facilitates metric selection and enables the combination of low-level measurements into indicators of the system's security level. This approach though has certain limitations as well. First, by matching security metrics to security patterns the framework may not be able to detect vulnerabilities exterior to the latter. Furthermore, no specific metrics were proposed, since, as the author notes, no definitive catalogue of such metrics exists. Moreover, the framework only provides an indication of the correct operation of the implemented security functions. As such, the state of security objectives that have not been taken into account during development is not reflected in the measurements.

Nichols et al. [54] presented a metrics framework used to detect security vulnerabilities that belong in the OWASP Top-10 list. Additionally the author introduced the usage of a security scorecard to group these issues

together. Such an approach though doesn't provide an accurate overall security status of the system as it may miss issues that do not belong in the OWASP list. Furthermore, one could argue that such an approach doesn't constitute a framework of correlated metrics but rather a collection of separate ones.

Scandariato et al. [62], finally, worked towards a framework of metrics by analyzing security principles that are relevant to the purpose of unearthing security properties and proposing suitable metrics to measure them. However, there is vague separation between maintainability concerns affecting security and pure security issues. Additionally, the metrics proposed are not being implemented, or elaborated in detail, they mainly apply to the architecture and design phase, and the validity or completeness of the framework is not validated. Furthermore, as the authors note, automation of the metrics is needed since some of these require high level of expertise and high degree of manual work.

Chapter 3

Research Methodology

In this chapter information is provided regarding the method followed to implement the metric-based security model. At the beginning, the requirements that the framework should comply to are given, as well as its purpose. Afterwards, the goal, question, metric approach, which was applied to identify the metrics, is provided.

3.1 Requirements

As already explained in the introduction, the aim of the research conducted within this thesis, is to develop a framework used to assess the security level of a software system. Set conditions, to be adhered to, by the model, are:

1. **Repeatability**: The framework should be able to produce the same output provided the same input is given.
2. **Effectiveness**: The framework should provide an accurate and as complete as possible impression of the security status of the application.
3. **Efficiency**: The framework should minimize time and human resources needed to produce the assessment.
4. **Objectivity**: The assessment produced by the framework should accurately rank the security status of different systems.
5. **Ability to be used throughout the system's life-cycle**: The framework should be able to be used throughout, most of, the system's

life-cycle, namely from the development phase until the application's maintenance.

The model consists of automated metrics, as to satisfy these requirements. Advantages of using metrics were elaborated in detail in chapter 2.

The purpose of the Metric-based Security Model, as the framework is named, is to assist stakeholders in making informed decisions as to where to concentrate their efforts. The model should prove of use to multiple levels of an organization. For example, developers should be able to detect where the issues are located, project managers or product owners where and how to distribute available resources and upper management should get an indication of the security level of the system.

3.2 Goal-Question-Metric Approach

After the definition of the requirements that the framework, and hence the metrics, should comply to, the Goal-Question-Metric approach was followed, in order to identify relevant metrics . The Goal-Question-Metric approach is a top-down, goal-oriented software measurement methodology of establishing appropriate metrics, proposed by Basili[18].

The goal of our framework, as mentioned above, is to evaluate the security level of a software system from the system owners perspective (e.g. developers, product owners, managers).

In this study we focus on software vulnerabilities as a means to maliciously attack a software system, and hence breach its security. Thus, the metrics we will define aim at assessing the security level of a software system, based on the existence of software vulnerabilities on its source code and external dependencies. Table 3.1 provides the approach we followed to determine the metrics. The latter are explained in subsection 3.2.1.

3.2.1 Explanation of the metrics

The framework is consisted out of the four metrics identified on Table 3.1. Table 3.2 presents the metrics based on the area and severity of issues each metric detects.

The first two metrics (Acute and Latent Issues) detect vulnerabilities within the source code of the system and provide a rating based on the

Goal	Purpose Issue Object Perspective	Evaluate the security level of a software system from the system's owner perspective.
Question	Does the source code of the system contain vulnerabilities?	
Metric	Acute Issues metric	
	Latent Issues metric	
Question	Do any external dependencies of the system contain vulnerabilities?	
Metric	Vulnerabilities on dependencies metric	
Question	Are the third party dependencies updated frequently?	
Metric	Dependency Freshness metric [27]	

Table 3.1: A GQM approach to define the framework's metrics.

number of vulnerabilities detected. The difference between the two metrics is the severity of the vulnerabilities each one is assigned with.

The Acute Issues metric detects high severity violations of secure coding patterns. These violations require no or minimal knowledge of the application's internal structure and are related with issues such as injection vulnerabilities, unvalidated redirects, sensitive data exposure etc. The Latent Issues metric detects and provides a rating for medium severity violations, such as vulnerabilities for which another program should be written to incorporate references to mutable objects, non final fields, etc.

The other two metrics detect issues located in third-party dependencies of the system. The Dependency Vulnerability metric, as the name states, detects vulnerabilities located on a system's dependencies and returns a rating based on the number of vulnerabilities found and their severity. Finally, the Dependency Freshness metric, reports how recently the third party dependencies of a system were updated.

The first three metrics were developed in the context of this study. The dependency freshness metric, developed by Cox[27], was integrated to the metric-based Security model, to provide an as more holistic view, of the security status of the third-party components, as possible. Further elaboration on the metrics will be given on Chapters 4, 5 and 6.

	Acute Issues.	Latent Issues.
Source Code.	High-severity violations of secure coding patterns. **(Acute Issues metric)**	Medium-severity violations of secure coding patterns. **(Latent Issues metric)**
External Dependencies.	Dependencies on components with known vulnerabilities. **(Dependency Vulnerability metric)**	Update profile of dependencies. **(Dependency Freshness metric)**

Table 3.2: Metrics based on area and severity.

Chapter 4

Framework Design

In this chapter a high level architecture of the framework is given, followed by a first explanation of the rating method of the metrics. Furthermore, the dataset of open source systems that is used in order to develop and test the framework is presented.

4.1 Architecture of the framework

The architecture of the framework is presented in Figure 4.1. The architecture of the system is divided in four levels (Input, Tools, Metrics Code, and Output) and in two main components ("Source code Metrics", and "External Dependencies Metrics").

The source code metrics take as input the source code of the system, which is checked for vulnerabilities by a static analysis tool (FindBugs[5]) and a plugin of it (Find Security Bugs [4]). Additionally, information regarding the size of the system is extracted with the use of SAT (Section 5.1.2), an internal tool to the organization in which the research was conducted. The Vulnerabilities on Dependencies metric, on the other hand, takes as input the third party dependencies of the system. Dependency Check [3] by OWASP is the tool which is used to report known vulnerabilities existing on these dependencies.

In the Metrics Code level, the output of the tools is processed to extract and process relevant information, rate the security status of the system, in regard to the focus of each metric, and present the ratings and other findings. Furthermore, the ratings of the metrics are aggregated in order to

Range	Stars
[0.5-1.5)	1 Star
[1.5-2.5)	2 Stars
[2.5-3.5)	3 Stars
[3.5-4.5)	4 Stars
[4.5-5.5)	5 Stars

Table 4.1: Metrics' star ratings

acquire the overall rating of the system.

Further elaboration on the tools used, the information processing methods, and the rating calculation and aggregation methods will be given on Chapters 5 and 6.

4.2 Metric Ratings

The rating generated by each metric shows the status of the system, in regard to the security characteristic analyzed by the metric. The rating is computed in a continuous scale from 0.5 to 5.5. This value can be converted into a natural value of stars by standard, round half up, arithmetic rounding (except for the limit value of 5.5 which is converted to 5 stars), as presented in Table 4.1. The notion of the star rating was adopted from the existing rating method used in SIG [10].

The rating produced by each metric is a relative rating, indicating the security status of the system when compared to other systems scanned. Furthermore the systems are assigned in the star categories by a predetermined distribution (e.g 5%-30%-30%-30%-5%). In order to achieve this, calibrated thresholds of each star category are used.

We use a relative rating, indicating the system's security performance in comparison to other systems, combined with an assignment of systems to the star categories according to a predefined distribution. We decided to follow this approach, since, we believe that, the result produced is easy to explain and interpret, is representative of real systems, which allows comparison and ranking, and captures enough information to enable traceability to individual measurements, allowing to pinpoint problems [10].

The exact methods of the rating calculation and benchmarking will be further explained in the relevant chapters (Chapters 5 & 6), since variations exist on each metric.

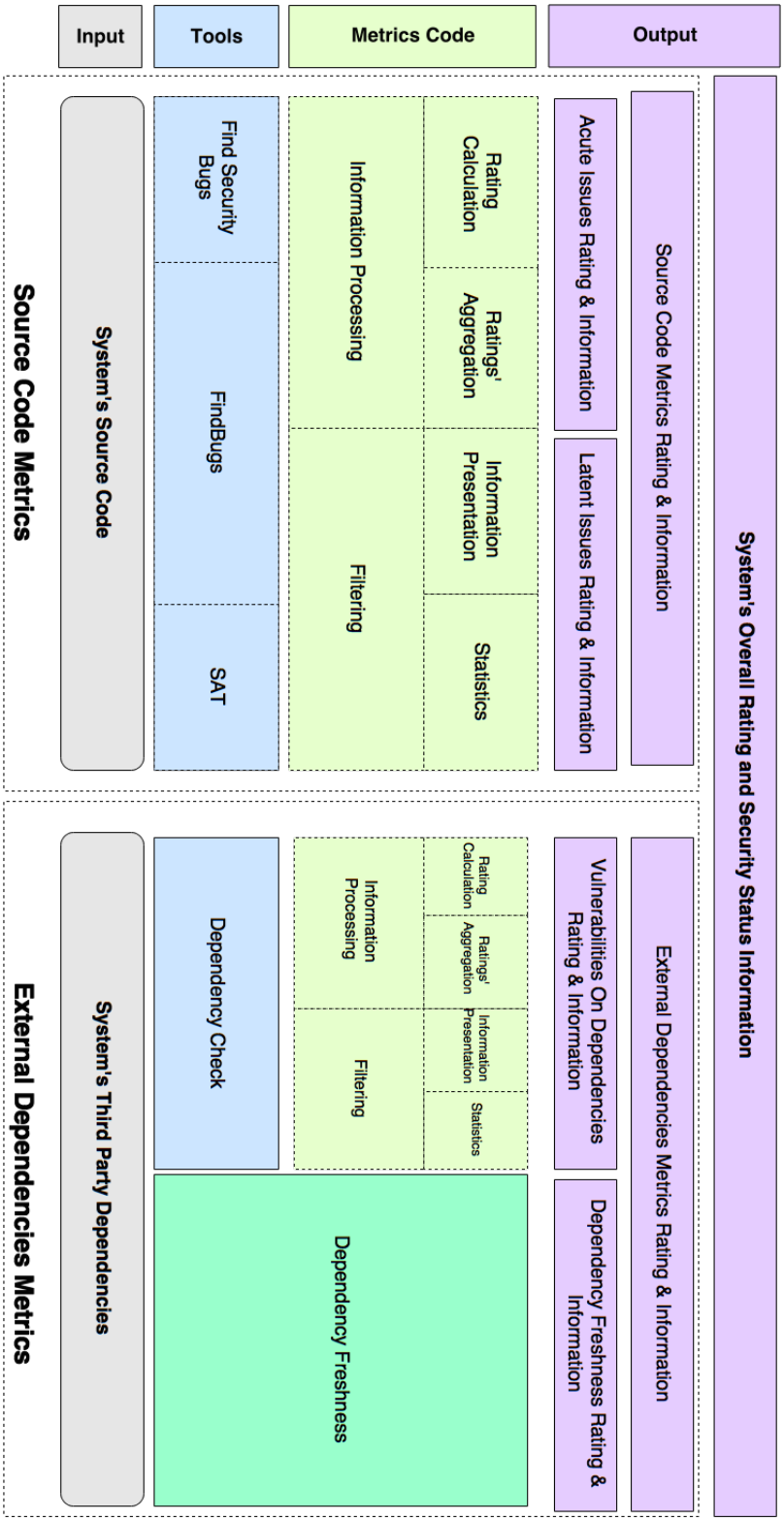


Figure 4.1: Architecture of Metric Based Security Model

4.2.1 Ratings aggregation method

The ratings produced by the four metrics of the model are aggregated in order to determine the system's overall rating. We decided to use the power mean as an aggregating approach (Equation 4.1). The power mean outclasses the simple arithmetic mean in the sense that it better reflects the weakest link principle. At the same time, though, it is more distinctive than simply using the minimum function for aggregation, as in that case, every system with a single serious flaw, would be treated the same. The same approach was used by Xu et al. [82], in their Security Model.

$$M_p(a_1, a_2, ..., a_n) \equiv (\frac{1}{n} \sum_{k=1}^{n} a_k^p)^{\frac{1}{p}} \tag{4.1}$$

4.3 Dataset

In order to develop the framework and calibrate the star ratings' thresholds, we used a dataset of 27 systems, 25 of which are open-source systems and 2 are SIG's internal systems (Table 4.2).

All the systems are developed in Java and use Apache Maven [1] to manage their build. Both conditions were a prerequisite when creating the dataset, since the model, at its current state, can evaluate, completely, only such systems. FindBugs, as will be explained in the next chapter, analyzes only Java systems, whereas Dependency Freshness requires the .pom file used by Maven to build a system. A Project Object Model (POM) file provides all the configuration for a single project, such as the project's name, its owner, dependencies on other projects, used plugins, etc.

It should be noted, here, that when creating the dataset, the context within the systems are used, and their type, was not taken into account. The Metric-based Security Model, detects vulnerabilities and rates systems accordingly, but does not verify whether the reported vulnerability can be exploited, by the current use of the system.

System Name	Description
Elastic Search	Search Engine.
Ninja	Web Framework.
Clojure	Programming Language.
H2O	Statistical, machine learning & math runtime for Big Data.
Fluent-Http	Web Server.
MapDB	Database Engine.
Jetserver	Game server.
KeyBox	Web-based SSH administration console.
Okhttp	Http Client.
Guacamole-client	Remote Desktop Gateway.
Libgdx	Game Development Framework.
HBC	Http Client.
OrientDB	Database Management System.
Oryx	Machine learning infrastructure.
Webbit	Websocket & Http server.
CAS	Single sign-on service.
Webmagic	Web crawler framework
WebGoat	Deliberately Insecure Web Server
Mahout	Machine learning applications framework.
AWS SDK	Amazon Web Services Software Development Kit
Guice	Dependency Injection Framework
WebScarab	Analysis Framework
Jenkins	Continuous Integration Server
McMMO	Game Modifications
Java-Websocket	Websocket client & Server
Internal System 1	
Internal System 2	

Table 4.2: Dataset Used.

Chapter 5

Source Code Metrics

The purpose of the source code metrics is to identify vulnerabilities, located in the source code of a software system, and provide a rating, based on these findings.

We detect the vulnerabilities through the use of a static analysis tool, FindBugs, and an extension of it. The advantages of using static analysis, compared to other forms, such as dynamic analysis, have already been presented in the Background chapter (chapter 2). The former, namely, allows the detection of security issues to occur earlier in the lifecycle of the system, since it doesn't require an executable version of the system ([47],[17]). Furthermore, static analysis can locate the source of a problem rather than the place where its symptoms occur [17]. Moreover, as Antunes [12], notes, static analysis tools usually have larger coverage than dynamic analysis tools.

After the detection of the vulnerabilities, the output of the tools is processed. The separation of the security issues between the metrics is based on their severity and type. The metrics' rating method, also, differs, taking into account their separate characteristics. Further information will be presented in the relevant sections.

This chapter's structure follows the layered logic of the model's architecture (Figure 4.1). At the beginning, information regarding the static analysis tools used, their operation and their output, is provided. Afterwards, the implementation of the metrics' layer is presented, elaborating on the relevant, to each metric, bug patterns, the rating methods used and the derived thresholds. Finally, the results of the application of the metrics, on the dataset, are displayed.

5.1 Tools

In the context of this research, a number of static analysis tools have been tested, such as SonarQube [7], PMD [6] and Yasca [8]. Currently, the acute and latent issues metrics use FindBugs and "Find Security Bugs" to detect vulnerabilities and SAT to acquire the size of the system's files.

5.1.1 FindBugs & Find Security Bugs

FindBugs is an open source static analysis tool for Java, that relies primarily in intraprocedural analysis to find known patterns of defective code [14]. Therefore, the tool uses analyzers called Bug Detectors to search for bug patterns [34]. Various heuristics are contained in these detectors to filter out or de-prioritize warnings that may be inaccurate, or that may not represent serious problems. FindBugs doesn't aim to be sound but rather tries to filter out warnings that may be incorrect or low impact [14]. Extensive research has been conducted in order to evaluate findBugs' usefulness and accuracy ([14],[76],[15],[77]), improve its reporting methods, in order to reduce the effect of false positives, ([64],[40],[41]), or evaluate other objectives through the use of this tool (e.g [53]).

The decision to use FindBugs lies in the following facts:

- **Open Source**: FindBugs is an open source tool making it easy to acquire and use for research purposes.
- **Fast**: When compared to other, more complicated, static analysis tools, FindBugs can analyze a system relatively fast, making it easier to scan a larger number of systems, as done in this research.
- **Extensive use**: FindBugs is used in a large scale by developers [26], when compared with other static analysis tools, and has, also, been used, extensively, for research purposes.

Nonetheless, it should be noted, that, FindBugs was used as a proof of concept. The framework doesn't eliminate the use of other tools, rather it supports such an idea. This statement, will be further elaborated in the Discussion chapter (chapter 8).

The tool analyzes compiled bytecode (.class or .jar files) of the system. Furthermore, it can be run in various modes, namely, from the command line, as was done within this research, as a stand-alone application, or as a plugin for several popular IDEs and continuous build systems.

Category	Description
Bad Practice	Issues that violate recommended coding practices.
Dodgy	Code that is confusing anomalous or error prone.
Performance	Issues affecting performance, such as inefficient memory usage/buffer allocation, usage of non static classes, etc.
Internationalization	Issues originating from use of non-localized methods.
Malicious Code	Issues originating from variables or fields exposed to classes that should not be using them.
Correctness	Apparent coding mistakes.
Multithreaded Correctness	Thread synchronization issues.
Security	Direct security related bug patterns.
Experimental	Experimental bug detectors.

Table 5.1: FindBugs Bug Categorization.

FindBugs warnings are grouped into over 380 bug patterns which in turn are grouped into 9 categories (Table 5.1). Out of the nine categories, we consider the Malicious Code and Security categories as the ones containing security related bug patterns. Therefore, the source code metrics only take into account these two categories. The same deduction was used by Mitropoulos et al. [53]. The output of FindBugs was chosen to be a .xml file containing the type, description and location of the reported vulnerabilities, to ease further processing.

"Find Security Bugs" plugin was used in combination with FingBugs. "Find Security Bugs" detects additional security vulnerabilities in web applications. The plugin can identify 63 different vulnerability types, which are integrated in the Security category.

5.1.2 "SAT"

The System Analysis Toolkit ("SAT") is a source code analysis tool developed and used by the Software Improvement Group ("SIG"). SAT's main purpose is to evaluate the maintainability of a software system. In that respect, it identifies areas needing attention and produces a benchmarked star rating of the system's maintainability and its relevant subcharacteristics as defined in ISO-IEC 25010 [38].

Within the context of the acute and latent issues metrics, we use SAT to acquire the files size of the systems analyzed.

5.2 Metrics

In the second layer of the model's architecture, the tools' output is processed in order to acquire the metrics' rating and additional information. At the beginning, the relevant data is extracted by FindBugs and SAT's reports, i.e the type and location of the vulnerabilities and the size of the files in which they belong. Afterwards, the vulnerabilities are split in the suitable metrics and the rating of the system for both metrics is calculated. Finally, all relevant information (such as rating, and location of vulnerabilities) are presented.

5.2.1 Identification of relevant Bug patterns to each metric.

As already mentioned, the vulnerability patterns considered relevant, belong in the Malicious code and Security categories. With the addition of those introduced by "Find Security Bugs", 89 patterns can be detected.

We decided to split the security bugs in the two metrics, depending on their type and severity level. Mitropoulos et al. [53] used FindBugs to study specific characteristics of security bugs individually and in relation to other software bugs. In his study, a similar approach was followed, by defining related to security bugs, the ones that belong in the Security and Malicious Code categories. Furthermore, the author divided security bugs in two severity categories, "Security Low" and "Security High".

The "Security High" category contained bugs "related to vulnerabilities caused from lack of user-input validation". Vulnerabilities of this type include attacks such as Injection (number 1 in OWASP top 10 [57]) and Cross-site Scripting (number 3 on OWASP top 10). As the author, of the paper, states, no knowledge about the application's internal structure is required for the exploitation of these vulnerabilities.

In the definition of the metrics, we adopted the introduction of these two categories, matching "Security High" to the Acute Issues metric and "Security Low" to the Latent Issues metric. Additionally we conducted some adjustments. First, the vulnerability patterns of the "Find Security Bugs" plugin were integrated. Furthermore, we decided to use OWASP's Top 10 list as a standard to extend the bug patterns included in the acute issues metric.

Specifically, the latter metric contains bugs belonging to the following entries of the list:

1. Injection (number 1).

2. Broken Authentication and Session Management (number 2).

3. Cross-Site Scripting (XSS) (number 3).

4. Insecure Direct Object References (number 4).

5. Sensitive Data Exposure (number 6).

6. Missing Function Level Access Control (number 7).

7. Unvalidated Redirects and Forwards (number 10).

The acute issues metric is a superset of the "Security High" category. Out of the ten entries of the OWASP list we cover seven with a total of 48 vulnerability patterns.

The Latent Issues metric assesses medium severity violations of secure coding patterns. In this metric belong vulnerabilities, not included in the OWASP Top 10 list, which require the development of an additional program to incorporate references to objects, fields, etc, or, have, by definition of the vulnerability pattern, a low true positive rate.

An example of a bug pattern belonging in the Latent Issues metric is the EI_EXPOSE_REP bug pattern (Table A.1). This pattern reports cases where an object's internal representation may be exposed by returning a reference to a to a mutable object's value stored in one of the object's fields. If instances are accessed by untrusted code, unchecked changes to the mutable object could compromise security or other important properties. Returning a new copy of the object is better approach in many situations.

An example of a bug pattern belonging in the Acute Issues metric is the SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE pattern (Table A.2). The method where this vulnerability is located invokes the execute or addBatch method on an SQL statement with a String that seems to be dynamically generated. A prepared statement should be considered used instead, since it is more efficient and less vulnerable to SQL injection attacks. This bug pattern corresponds to the Injection entry of the OWASP Top 10 list.

A presentation of the vulnerability patterns used, their corresponding metric and the OWASP Top 10 entry, in which they belong, is given in Appendix A.

5.2.2 Statistical Analysis

After the allocation of the vulnerability patterns to the metrics, and the generic definition of the rating method to be used (section 4.2), relevant statistical information of the dataset, in respect to the two metrics, was gathered. This was done for the following reasons:

1. To gain knowledge on some of the dataset's properties, such as how many bugs exist, how many distinct bug patterns are reported, etc.
2. Statistical information could help in the detailed definition of the rating methods to be used and the calculation of the star categories thresholds. Relevant values necessary to define the thresholds are the number of bugs per file of the system, their minimum and maximum values, etc.
3. To test whether the number of bugs per system, for both metrics, correlates with the system's size. Mitropoulos et al. [53], in his study found that the Spearman correlation of the Security Low and Security High categories was 0.65 and 0.19 respectively. In case their findings were confirmed in our dataset as well, this would indicate that there is a strong connection of, mainly, the latent issues metric with the size of the system. As a result, the calculation of the rating should take into account the size of the system, in such a way that larger systems can be treated equally to smaller ones.

The metrics only evaluate the security level of normal files and not test files. This was a deliberate decision since security issues in test files are of less significance. The statistical analysis conducted therefore, only refers to normal files.

At the beginning the number of potential vulnerabilities, as well as the number of different vulnerability patterns, found in the dataset was checked. In total 3,322 bugs were found. Out of these, 3,128 (94.16%) belong in the "Latent issues" metric and 194 (5.6%) belong in the "Acute issues" metric (Figure 5.1).

Furthermore, out of the 89 vulnerability patterns contained in the metrics, 39 were present in the dataset, of which 13 belong in the "Latent Issues" metric and 26 in the "Acute Issues" metric.

The distribution of bugs per file was examined next (Table 5.2). The dataset contains a total of 24,981 files. Out of them 1,271 contain latent issues whereas 130 contain acute issues. The maximum number of latent and

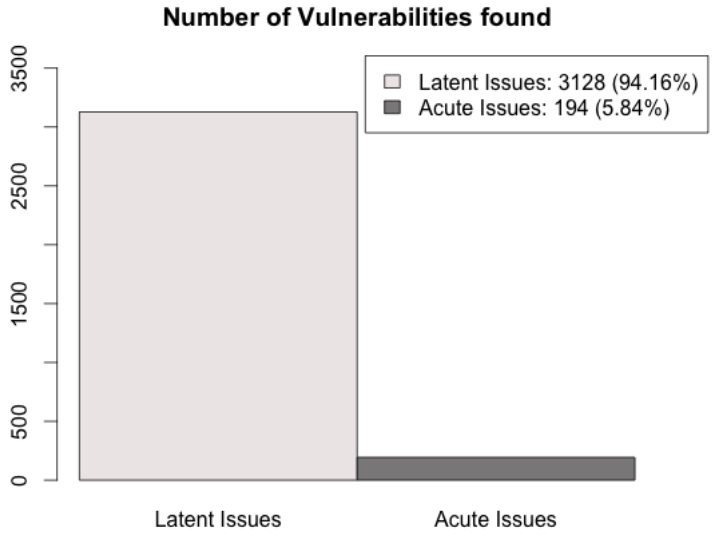


Figure 5.1: Number of vulnerabilities per metric.

All files	Minimum	Maximum	Median	Mean
Latent Issues	0	22	0	0.125
Acute Issues	0	7	0	0.08

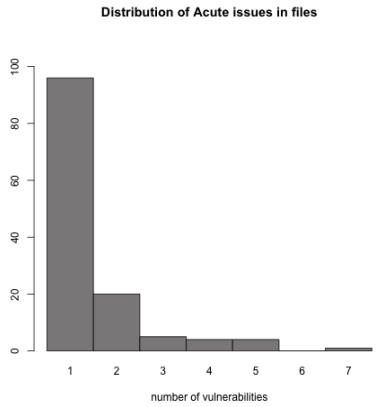
Table 5.2: Distribution of vulnerabilities in all files.

acute issues per file is 22 and 7 respectively, whereas their mean is 0.125 and 0.08.

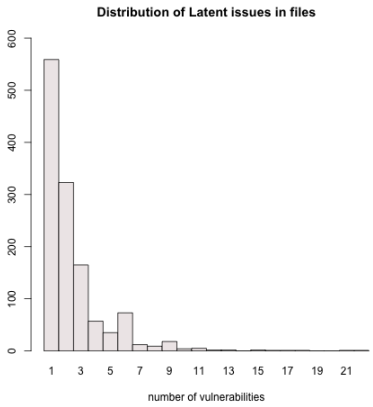
Additionally, the distribution of vulnerabilities on vulnerable files only (Table 5.3) was examined. A vulnerable file, concerning a metric, is defined as one that contains at least one reported vulnerability belonging in that metric. The median of latent issues per vulnerable file is 2 whereas the mean is 2.461. Furthermore, the median of acute issues is 1 whereas the mean is approximately 1.5. In figures 5.2a and 5.2b barplots of the distribution of the number of vulnerabilities in vulnerable files are presented.

Vulnerable files	Minimum	Maximum	Median	Mean
Latent Issues	1	22	2	2.461
Acute Issues	1	7	1	1.492

Table 5.3: Distribution of vulnerabilities in vulnerable files.



(a) Acute Issues.



(b) Latent Issues.

Figure 5.2: Distribution of issues in vulnerable files.

Size (loc)	Minimum	Maximum	Median	Mean
All files	2	6,984	42	79.211
Latent files	9	6,984	125	223.152
Acute files	13	2,412	150	247.062

Table 5.4: Size of files (in lines of code).

The size of the dataset's files and its relation with the presence of latent and acute issues was also examined. For all files, the minimum length is 2 lines of code (without comments and blanks) whereas the maximum is 6984. The median, additionally, is 42 and the mean is approx. 80 (Table 5.4).

Vulnerable files are in average larger than the average of all the files combined, with their mean being approximately 223 and 247 for files containing latent and acute issues, respectively. Figure 5.3 shows the distribution of files' size for the different types of files.

Finally the correlation of the systems' size with the number of vulnerabilities, for each metric, was also calculated. As Table 5.5 and Figures 5.4a and 5.4b illustrate this was found to be approximately 0.97 for the latent issues and 0.42 for the acute issues.

	Acute Issues	**Latent Issues**
Pearson Correlation	0.415	0.966
p values	0.03116	4.441×10^{-16}

Table 5.5: Correlation of Systems' size with number of vulnerabilities.

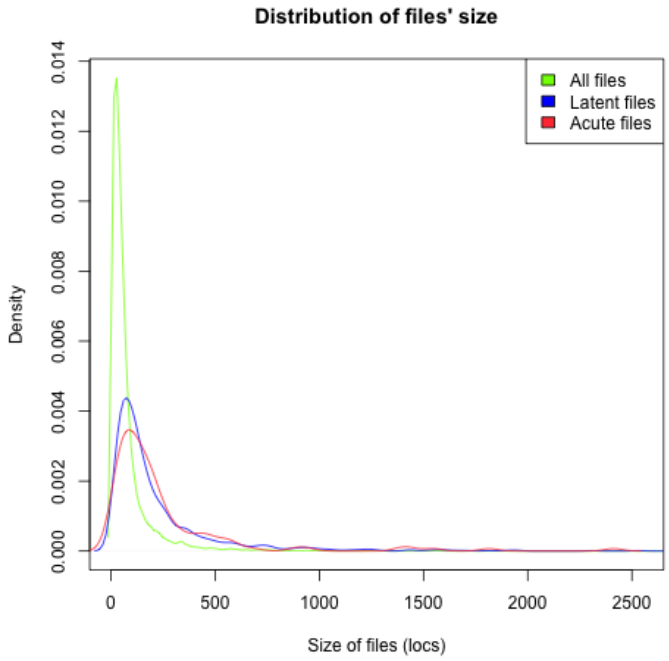
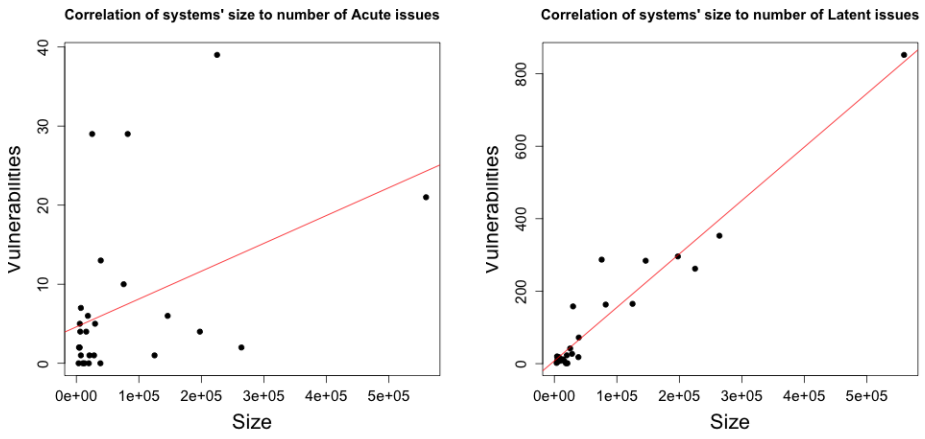


Figure 5.3: Distribution of files' size.



(a) Acute Issues.

(b) Latent Issues.

Figure 5.4: Correlation of systems' size to number of vulnerabilities.

5.2.3 Latent issues rating method

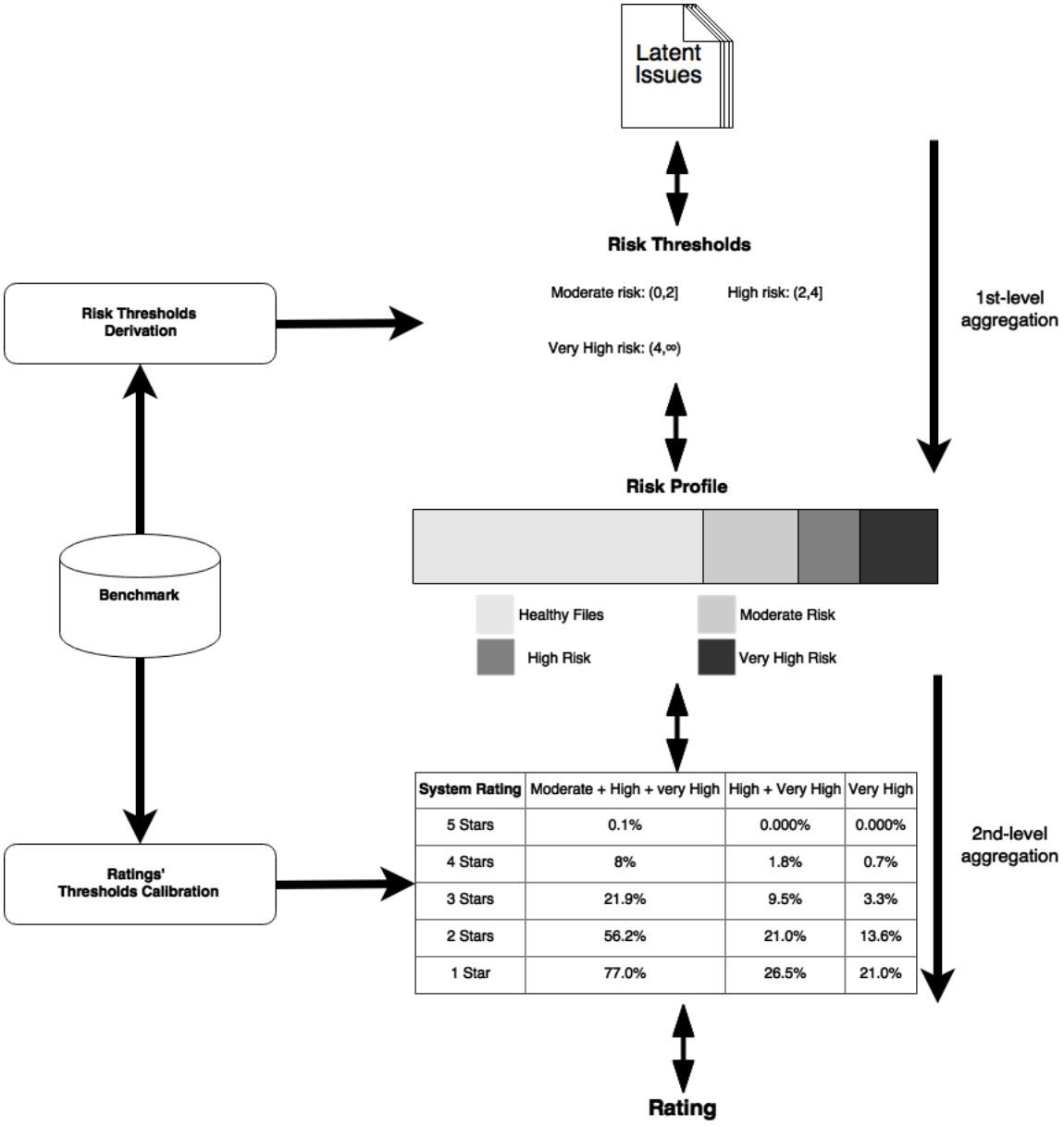
As already mentioned in section 4.2 the metrics' rating methods use benchmarked thresholds, in order to define the boundaries between the star categories. The advantage of this approach is that the result produced is easy to explain and interpret, is representative of real systems, which allows comparison and ranking, and captures enough information to enable traceability to individual measurements, allowing to pinpoint problems [10].

However the specific implementation of each rating method differs, based on what aspect each rating tries to emphasize as well as specific characteristics of each metric. The acute issues metric, for instance, gives emphasis on the number of vulnerabilities found, irrelevant of their distribution within files. On the other hand, the latent issues metric, doesn't calculate the rating based on the absolute number of vulnerabilities found, rather it takes into consideration how the issues spread, i.e. their density within the system. This rating implementation directly relates to the purpose of the latent issues metric; specifically to find and rate medium severity violations of secure coding practices, i.e. violations that indicate lack of quality coding (in respect to security) from the developers, resulting to possible security problems in the future. Furthermore, the rating method used in the latent issues metric takes into account the high correlation existing between the size of the systems and the number of latent issues. This approach is necessary, in order to treat equally differently sized systems.

For the calculation of the metric's rating we adopt the two-stage aggregation of metrics into ratings methodology, proposed by Alves et.al. [10]. Figure 5.5 presents an overview of the approach used to aggregate the measurements to ratings, using benchmarked-based thresholds, as well as the value of the thresholds that were defined. As illustrated in the picture, the aggregation of individual measurements to ratings is done through a two-level process.

First, individual measurements (number of latent issues for each file) are aggregated to risk profiles based on risk thresholds (1st level of aggregation). A risk threshold indicates the boundary value between the three risk categories we have defined:

- **Moderate**: Contains files with one or two latent issues.
- **High**: Contains files with three or four latent issues.
- **Very High**: Contains files with more than 4 latent issues.



System Rating	Moderate + High + very High	High + Very High	Very High
5 Stars	0.1%	0.000%	0.000%
4 Stars	8%	1.8%	0.7%
3 Stars	21.9%	9.5%	3.3%
2 Stars	56.2%	21.0%	13.6%
1 Star	77.0%	26.5%	21.0%

Figure 5.5: Two-level aggregation rating method.

A risk profile represents the percentage of overall code that falls into each of the three risk categories. For example, if a file contains three vulnerabilities, the size of the file (specifically, the ratio of the file's size to the overall size of the system) is added in the high risk category in the risk profile of the system.

After the calculation of the system's risk profile, the latter is aggregated into a rating by determining the minimum rating for which the size of the risk profile categories doesn't exceed a set of second level thresholds (2nd level of aggregation). Each set of thresholds is calibrated in order to achieve a 5%-30%-30%-30%-5% distribution of systems in the star categories. More information on how we acquired the values of the first and second level thresholds can be found on Appendix B.

A system's rating can be represented both in a discrete and a continuous scale. The discrete scale is achieved by comparing the values of the system's risk profile with the set of second level thresholds. The continuous scale is achieved by using an interpolation function between the lower and upper threshold of the star category the system belongs. The interpolation function is presented in Equation 5.1.

$$s(u) \;=\; s_0 \;+\; 0.5 \;-\; \frac{(u \;-\; t_0)}{t_1 \;-\; t_0} \tag{5.1}$$

Where $s(u)$ is the final rating, u the Percentage of volume in the risk profile, s_0 the initial discrete rating, t_0 the lower threshold for the risk profile, and t_1 the upper threshold for the risk profile

The final interpolated rating $R\,\epsilon\,[0.5\,,\,5.5]$ is then obtained by taking the minimum rating of all risk categories, as follows:

$$R \;=\; min\,(s(RP_{M+H+VH}), s(RP_{H+VH}), s(RP_{VH}))$$

The choice of range from 0.5 to 5.5 is so that the number of stars can be calculated by standard, round half up, arithmetic rounding. In the possible situation, though, that a system achieves a continuous rating of 5.5 (there are no latent issues in the system), the rating should be trancated rather than rounded.

Category	Thresholds	Proposed Distribution
5 Stars	0	-
4 Stars	(0,2]	30%
3 Stars	(2,6]	30%
2 Stars	(6,28]	30%
1 Star	(28,36]	10%

Table 5.6: Thresholds of "Acute issues" rating method.

5.2.4 Acute issues rating method

A different approach is followed to calculate the rating of the acute issues metric. Specifically, a single level aggregation, based on the number of the acute issues, is performed. Acute issues have different properties than the latent issues. For that reason, the number of vulnerabilities per file is not considered relevant, but rather we take into account only the absolute number of issues in the system. Furthermore, we found out, in the statistical analysis, that there is only low correlation between the number of acute issues and the size of the system. Therefore, normalization on the system size is not conducted.

In order to calibrate the thresholds the number of acute issues per system was identified for all the systems in our dataset, and then we calculated the minimum thresholds that can categorize those systems based on a given distribution.

This method is similar to the second level of aggregation of the latent issues rating method. However, in this case we decided to apply the distribution only on the four lower star categories, i.e the "5 Stars" category doesn't have a calibrated threshold, rather systems take five stars only if they don't have any acute issues. This decision was taken, in order to emphasize the increased severity that acute issues can have on a system.

Table 5.6 presents the derived thresholds, used on the aggregation of acute issues to a rating, as well as the defined distribution of systems to star categories. As is the case, also, on the latent issues rating method, the ratings can be represented both by an integral and a decimal number. Furthermore, for the calculation of the decimal rating the interpolation function of Figure 5.1 is used. Due to the fact, though, that we do not include the 5 star category in the defined distribution, a continuous rating for systems with 5 stars can not be calculated. Furthermore, systems that exceed the lower limit of the 1 star category receive a default continuous rating of 0.5 stars.

5.3 Results

The results section is divided in two subsections. The first presents an example of how a single system is processed by the metrics, the calculation of its ratings and the output produced. The second subsection provides the results of applying the metrics to all the systems of the dataset, i.e. the ratings of the systems and the distribution of systems to the star categories of the metrics.

5.3.1 System-specific results

The results of Jenkins, a continuous integration server, will be presented as an example. The software consists of 964 files with total size of 82,148 lines of code (without blanks and comments). After scanning the system with FindBugs we identified 192 reported vulnerabilities, 163 of which are latent issues and 29 are acute issues. The latent issues were concentrated in 84 files, with total size of 23,759 lines of code, and 16 files of total size of 4,921 lines of code, contained acute vulnerabilities. Table 5.7 summarizes the findings.

	Overall	**Latent**	**Acute**
Vulnerabilities	192	163	29
Files	964	84	16
Size (in LOC)	82,148	23,759	4,921
Rating		2.029 (2 Stars)	1.375 (1 Star)

Table 5.7: Results of Jenkins.

In order to produce the rating of the latent issues metric, the system's risk profile was calculated (Figure 5.6). Approximately, 71% of the overall size of the system contains no vulnerabilities. Files consisting the 16,7% of the overall size of the system belong in the moderate risk category, whereas the other two categories, of high and very high risk, contain around 4% and 8,1%.

Next, Jenkin's cumulative risk profile was compared with the second level thresholds of Figure 5.5. Table 5.8 presents the system's cumulative risk profile that is directly compared with the thresholds.

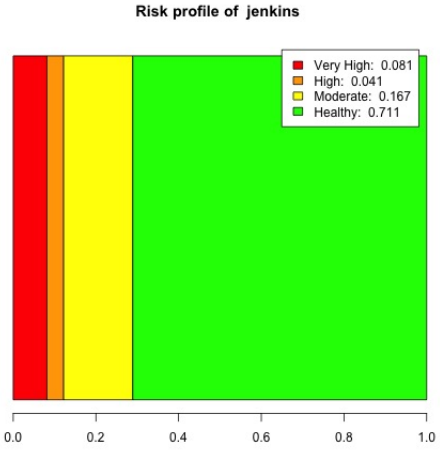


Figure 5.6: Risk profile of Jenkins.

	M + H + VH	**H + VH**	**VH**
Jenkins	0.289	0.122	0.081

Table 5.8: Cumulative risk profile.

We notice that the system belongs in the 2 Stars category. In order to calculate its continuous rating the metric uses the interpolaration function of Equation 5.1, with a result of 2.029. An example of the traceability provided by this rating method is given below. In case we wanted to increase the rating to 3 stars we would have to reduce the Very High risk category by at least 4.8%. and the cummulative size of all the risk categories by at least 7%. By noticing the distribution of reported vulnerabilities in the files, we notice that we would have to correct issues in two files of the Very high risk category with cummulative size of 4,237 lines of code. Furthermore, in order to reduce the cummulative size of all the risk categories by at least 7% we could focus on correcting issues in just four files with cummulative size above 7% of the overall size of the system.

The calculation of the acute issues rating is simpler. Jenkins contains 29 issues. Based on Table 5.6 it belongs in the 1 Star category. The interpolaration function of Equation 5.1 is used to calculate its continuous rating.

The output of the metrics, additionally, includes information regarding the location of the vulnerabilities in the system, their type, and their distribution between files, in order to prioritize the files needing correction.

5.3.2 Results from the dataset

The ratings produced, from the application of the metrics on all the systems of the dataset, are given on Table 5.9. The distributions of the systems in the star categories are presented in Figure 5.7 and on Table 5.10. As we can notice the distributions approach the defined distributions (in percentages) of 5%-30%-30%-30%-10% and 30%-30%-30%-10% for the latent and acute issues (without the five stars category) respectively. Any variations from the defined distributions are caused due to the small size of our dataset and the few unique values of acute vulnerabilities existing, making it impossible to achieve a perfectly aligned distribution.

We additionally checked to verify whether the ratings produced by the Latent and Acute Issues metrics correlate with the size of the systems (Table 5.11 and Figure 5.8a & 5.8b). As a result, of the rating methods used, there seems to be no significant correlation between the ratings produced by the metrics and the size of the systems (-0.053 for the Latent issues and -0.377 for the Acute). Furthermore, as seen in Table 5.11 and Figure 5.9 there seems to be no significant correlation between the ratings of the two metrics, as well (0.144).

System	Latent Issues	Acute Issues
AWS SDK	2.831	1.818
CAS	4.684	2.5
Clojure	1.746	5.5
Elastic Search	3.185	3.5
Fluent-Http	3.069	4.0
Guacamole-client	4.036	3.0
Guice	4.429	4.0
H2O	0.506	2.318
HBC	4.399	5.5
Jetserver	4.046	5.5
Java-Websocket	0.5	3.5
Jenkins	2.029	1.375
KeyBox	2.196	3.5
Libgdx	2.499	3.0
Mahout	3.101	4.0
MapDB	1.618	5.5
McMMO	2.059	2.75
Ninja	3.540	5.5
Okhttp	4.029	3.0
OrientDB	2.433	2.5
Oryx	3.510	4.0
WebScarab	2.535	2.181
Webbit	2.498	3.5
WebGoat	2.499	1.375
Webmagic	2.940	2.454
Internal System 1	- *Confidential*	- *Confidential*
Internal System 2	- *Confidential*	- *Confidential*

Table 5.9: Ratings of systems.

Stars	Latent (in %)	Acute (in %)	Acute for four Categories (in %)
5 Stars	3.7	18.5	-
4 Stars	25.9	29.6	36.4
3 Stars	29.6	25.9	31.8
2 Stars	33.3	14.8	18.2
1 Star	7.4	11.1	13.6

Table 5.10: Distribution of systems in Star Categories.

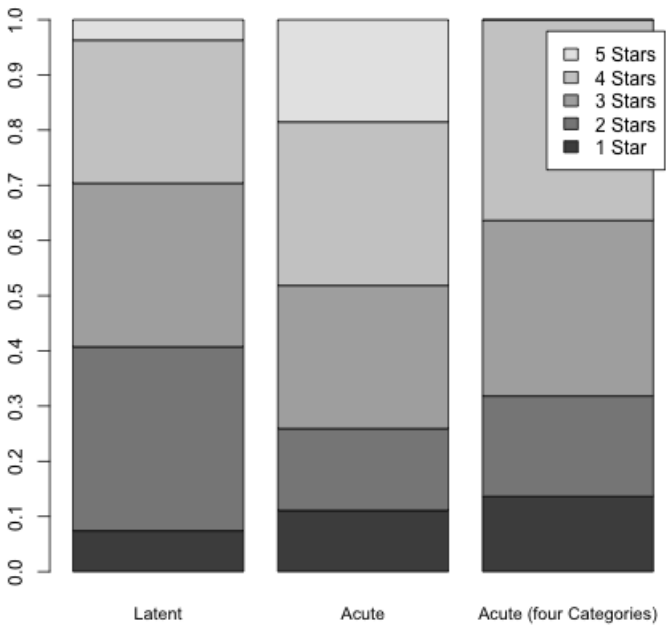
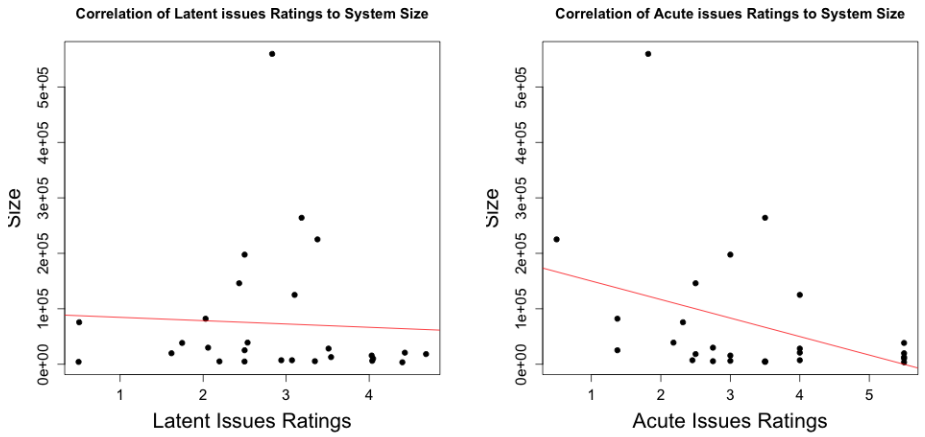


Figure 5.7: Bar plot of distribution of systems.

	Systems Size (loc)	**p-value**	**Acute Issues**	**p-value**
Latent Issues	-0.053	0.79	0.144	0.47
Acute Issues	-0.377	0.052	-	-

Table 5.11: Correlations of Metrics' ratings.



(a) Correlation of Latent Issues ratings to system size.

(b) Correlation of Acute Issues ratings to system size.

Figure 5.8: Correlation of metrics' ratings to system size.

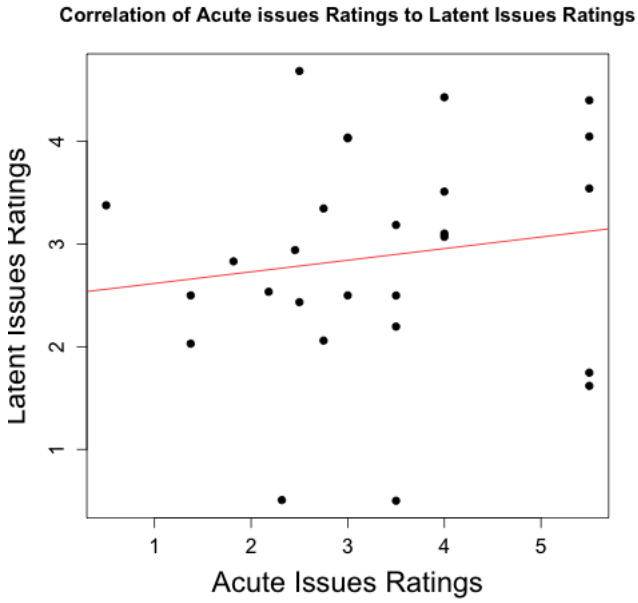


Figure 5.9: Correlation of Latent Issues ratings to Acute Issues ratings.

Chapter 6

External Dependencies Metrics

In modern software development, systems consist of multiple components, which can be developed independently of one another. Components have been described as the Lego blocks of software engineering [22]. More formally, Taylor, Medvidovic & Dashofy[70] define a software component as an architectural entity that encapsulates a subset of the system's functionality and/or data, restricts access to that subset via an explicitly defined interface, and has explicitly defined dependencies on its required execution context.

Components developed by third party organizations, either proprietary or open-source, are often integrated in software systems, providing benefits on cost, time and even performance. However, vulnerabilities contained on these components can endanger the security of the whole system. The OWASP Top 10 list, as mentioned in the background chapter (chapter 2.1.2), draws attention to this problem, in its 9th entry, "Using components with known vulnerabilities".

The metrics described in this chapter, aim at discovering vulnerabilities on these external dependencies of a system (section 6.1) or discovering non up-to-date external dependencies which could result in security issues (section 6.2). Additionally, the results of the application of these metrics on the dataset is given in section 6.3.

6.1 "Dependency Vulnerability" metric

The "Dependency Vulnerability" metric aims at identifying known vulnerabilities on the external dependencies of the system. At the beginning of this section, the tool that is used to identify these vulnerabilities is described. Afterwards, the processing of the output of the tool, as well as the rating method applied on the metric, are presented.

6.1.1 Tool

OWASP's Dependency-check [3] is used to identify external dependencies with known vulnerabilities. The tool can currently be used to scan Java, .NET, and Python applications to identify known vulnerable components. In addition, Dependency-check can be used to scan some source code, including OpenSSL source code and source code for projects that use Autoconf. In the context of this thesis, we use it to scan the external dependencies (in the form of .jar files) of the open-source Java systems contained in our dataset.

Dependency-check uses analyzers to collect information about the files it scans. The information collected is called evidence; there are three types of evidence collected: vendor, product, and version. For instance, the JarAnalyzer will collect information from the Manifest, pom.xml, and the package names within the .jar files scanned and by using heuristics it places the information gathered into one or more types of evidence.

The three types of evidence collected, originate from the Common Platform Enumeration (CPE) naming scheme [50]. CPE provides a standard machine-readable format for encoding names of IT products and platforms and includes a formal name format, a method for checking names against a system, and a description format for binding text and tests to a name. A Uniform Resource Identifier binding of the CPE follows the format "$cpe : / [EntryType] : [\mathbf{Vendor}] : [\mathbf{Product}] : [\mathbf{Version}] : [Revision] : \cdots$".

The National Vulnerability Database's (NVD) [55] Common Vulnerabilities and Exposures (CVE) Database [51] uses the CPE naming scheme to report software which contains vulnerabilities. An example of a CVE entry is given in Figure 6.1. Dependency-check uses the evidence collected by its analyzers and attempts to match the entries with entries contained in the, locally downloaded by the tool, CVE database.

The tool uses confidence levels to rate the evidence found - low, medium, high and highest. These confidence levels are applied to each item of evi-

```
<entry id="CVE-2012-5055">
...
  <vuln:vulnerable-software-list>
    <vuln:product>cpe:/a:vmware:springsource_spring_security:3.1.2</vuln:product>
    <vuln:product>cpe:/a:vmware:springsource_spring_security:2.0.4</vuln:product>
    <vuln:product>cpe:/a:vmware:springsource_spring_security:3.0.1</vuln:product>
```

Figure 6.1: Example of a Common Vulnerabilities and Exposures entry.

dence. When the CPE is determined it is given a confidence level that is equal to the lowest confidence level of evidence used during identification. If only highest confidence evidence was used in determining the CPE then the CPE would have a highest confidence level.

Output of the tool

We configured Dependency-check to output a .xml report of the findings produced, in order to facilitate easier processing on the metric layer. The top of the report contains information regarding the version of the CVE database that is used. Afterwards, general information is given, about the scan conducted, such as the name of the system and the date of the scan. The main body of the report presents the dependencies found on the system. Among the information provided is the name of the file of the external dependency, the directory where the file was found, the related dependencies found in the scan, the maven identifier of the dependency (if maven was used), the evidence that were collected and possibly a description of the dependency, if existing.

Furthermore, if the dependency is found to contain vulnerabilities additional entries are provided, such as the CPE identifier of the dependency, the confidence level of the CPE identifier, and an enumeration of the known vulnerabilities existing. This enumeration contains, for each vulnerability, the CVE identifier of the vulnerability, its CVSS score[1], references where the issue is reported, the CWE identifier of the issue, all the software vulnerable to this vulnerability, as well as a description of the vulnerability.

6.1.2 Metric

After the scan of the software system with Dependency-check is complete, the output of the tool is processed in order to produce the final rating of the

[1]The Common Vulnerability Scoring System (CVSS) is an open framework for communicating the characteristics and severity of software vulnerabilities [2].

system. First, the needed information, for the dependencies of the system, is extracted. Afterwards, simple filters we developed, try to reduce the effect of false positives and inaccurate confidence levels of CPE identifiers. Finally, the system is rated, based on the severity and number of the vulnerabilities identified.

Processing of tool output

We produce two tables from the report of Dependency-check. The first table contains the filename of all the dependencies found on the system, as well as the number of related dependencies of each dependency. An example of the first table is given in figure 6.2.

The second table contains information regarding the dependencies that were found to contain known vulnerabilities. The fields of the table contain the filename of the dependency, the number of related dependencies, its maven identifier if existing, its CPE identifier, the confidence level of the CPE identifier, the CVE identifier of the vulnerability and its CVSS score. An example of the second table is given in figure 6.3.

Due to how dependency-check identifies libraries, false positives may occur (a CPE was identified that is incorrect). Furthermore, the dependency that was identified may be used for testing purposes and hence not be relevant for the rating of the system. For that reason, we developed some basic filters to try and increase the precision of the confidence levels produced, based on observations we made during manual inspection of the tool's reports. It should be noted though, that these filters were developed as a proof of concept and hence further research on their effectiveness should be done.

The first filter excludes dependencies located in directories of the system used for testing purposes or whose filename suggests that they are used for testing. The second filter reduces the confidence of the HintAnalyzer [2] of Dependency-check in case the evidence of the analyzer appear not to be related to the dependency's filename or increases it if they are. The third filter, reduces the confidence level of CPE identifiers that do not appear to share any common words with the maven identifiers of the dependency. Finally, the fourth filter increases the confidence of specific CPE identifiers that were observed to be most of the times true positives but had a low confidence level. At the moment the fourth filter identifies only one such

[2]The HintAnalyzer uses knowledge about a dependency to add additional information to help in identification of identifiers or vulnerabilities. At the moment the hint analyzer identifies dependencies contained in the Spring Framework.

	Dependency	Number_of_related_dependencies
1	aws-java-sdk-autoscaling-1.9.18.jar	4
2	commons-codec-1.6.jar	46
3	commons-logging-1.1.3.jar	45
4	httpclient-4.3.4.jar	46
5	httpcore-4.3.2.jar	46
6	jackson-annotations-2.3.0.jar	46
7	jackson-core-2.3.2.jar	46
8	jackson-databind-2.3.2.jar	46

Figure 6.2: Example of the table containing the dependencies of a system.

	Dependency	Related Dependencies	Maven Identifier	CPE Identifier	Confidence	Vulnerability	Severity
1	httpclient-4.3.4.jar	46	(org.apache.httpcomponents:httpclient:4.3.4)	(cpe:/a:apache:httpclient:4.3.4)	HIGHEST	CVE-2014-3577	5.8
2	original-aws-java-sdk-osgi-1.9.18.jar	1	(com.amazonaws:aws-java-sdk-osgi:1.9.18)	(cpe:/a:springsource:spring_framework:1.9.18)	LOW	CVE-2014-1904	4.3
3	original-aws-java-sdk-osgi-1.9.18.jar	1	(com.amazonaws:aws-java-sdk-osgi:1.9.18)	(cpe:/a:springsource:spring_framework:1.9.18)	LOW	CVE-2014-0054	6.8
4	original-aws-java-sdk-osgi-1.9.18.jar	1	(com.amazonaws:aws-java-sdk-osgi:1.9.18)	(cpe:/a:springsource:spring_framework:1.9.18)	LOW	CVE-2013-7315	6.8
5	original-aws-java-sdk-osgi-1.9.18.jar	1	(com.amazonaws:aws-java-sdk-osgi:1.9.18)	(cpe:/a:springsource:spring_framework:1.9.18)	LOW	CVE-2013-6429	6.8
6	original-aws-java-sdk-osgi-1.9.18.jar	1	(com.amazonaws:aws-java-sdk-osgi:1.9.18)	(cpe:/a:springsource:spring_framework:1.9.18)	LOW	CVE-2013-4152	6.8

Figure 6.3: Example of the table containing the vulnerable dependencies of a system.

external dependency but more can be added through observation.

Effect of filters

In order to verify the precision[3] of Dependency-check and the effect of the developed filters, we conducted a small scale experiment on our dataset of systems. More specifically, we manually checked the reported vulnerable dependencies of 9 open source systems to determine whether they were true or false positives and verify whether there is an improvement after the application of the filters. A complete table of the reported vulnerable dependencies together with the indication of whether these are true or false positives is given on Appendix C.

Table 6.1 presents the results of the experiment. We identified 63 filenames of dependencies. The precision of the tool for all CPE confidence levels is 65.08%. When we take into account only the high and highest confidence level the precision increases to 89.96%, containing 45.45% of the total true positives found. After the application of the filters, the precision for the high and highest confidence level increases to 96.77% with a recall of 73.17%.

	Precision	**Recall**
All confidence levels	65.08%	-
$\geq$ High confidence	89.96%	45.45%
$\geq$ High confidence *after filters*	96.77%	73.17%

Table 6.1: Precision of Dependency-check.

[3]As precision we define the ratio of $\frac{True\ positives}{All\ reported\ vulnerabilities}$

Rating Method

We use a single level aggregation of measures to ratings, similar to the one used for the calculation of the source code acute issues metric (section 5.2.4).

In order to produce a more accurate rating we decided to take into account during its calculation, only the dependencies whose CPE identifiers are of high and highest confidence. Furthermore, we do not rate the number of vulnerable dependencies found in the system, but rather the total number of individual[4] vulnerabilities contained in these vulnerable dependencies. We define three severity levels of vulnerabilities:

- **Medium**: a vulnerability whose CVSS score is ≤ 4.5.
- **High**: a vulnerability whose CVSS score is ≤ 5.9.
- **Very High**: a vulnerability whose CVSS score is > 5.9.

The defined thresholds are presented in table 6.2. The thresholds were acquired from additional research conducted within [21].

Category	**M+H+VH**	**H+VH**	**VH**
5 Stars	0	0	0
4 Stars	1	1	0
3 Stars	4	4	0
2 Stars	10	7	2
1 Star	14	10	2

Table 6.2: "Dependency Vulnerability" thresholds.

6.2 "Dependency Freshness" metric

Dependencies have to be updated to ensure the flexibility, security, and stability of the system. Software vulnerabilities, existing on external dependencies of the system, may be repaired in subsequent releases of the component. However, without proper updating of the external dependencies, these fixes will not be applied to the system using these dependencies. Indeed, Cox et. al. [27] found that systems with outdated dependencies are more than four times as likely to have security issues in their external dependencies.

[4]By "individual" we mean that in case a CVE of a vulnerability appears more than once on the system, we take it into account only the first time.

For that reason, the "Dependency Freshness" metric, aims at assessing the update profile of the third party components used in a system. In that direction we adopt the benchmarked-based metric developed by Cox et. al. [27].

The author, in his thesis, presented several measurements to quantify how outdated an individual dependency is, as well as a benchmark-based metric to rate a system as a whole. To determine which measurement method will be applied in order to quantify the update behavior of dependencies, he assessed several methods based on the criteria defined by Heitlager et.al.[32]. As a result, the version sequence number distance[5] was selected as the measurement. The author defended this choice by noting that this measurement discounts dependencies on quick release cycles (which are thus more subject to change), rather than mature projects which only see minor updates.

The metric produces a benchmarked star rating using the same approach as the latent issues metric applied in the source code of the system (section 5.2.3). The distribution of systems in the star rating categories aims at being at 5%-30%-30%-30%-5%. A dataset of 75 systems was used to develop and test the metric.

The latter, in its current implementation, assesses Java systems which manage their dependencies through Maven. To determine the dependencies of a system and their exact versions, the manifest file (pom.xml) of the system is being read. Furthermore, to complete the database of dependency versions, and be able to determine the version sequence distance, the Maven.org repository is used.

6.3 Results

In this section the results from the application of the metrics on the dataset's systems will be provided. At the beginning the ratings and additional output of each metric is given. Afterwards, we examine whether there is any correlation between the ratings produced by all four metrics of the framework. Finally, we check whether there, additionally, exists any correlation between the ratings produced by our metrics and ratings assessing maintainability properties of the dataset's systems.

[5]The version sequence number distance is defined as the difference of the version sequence numbers of two releases.

6.3.1 "Dependency Vulnerability" metric results

The ratings of the systems belonging in our dataset are given in Table 6.3. The metric, additionally, produces, for each system, the tables presented in Figures 6.2 and 6.3. The aim of these tables, is to assist systems' stakeholders to identify all the external dependencies that are used in the system, as well as the ones containing known vulnerabilities.

The distribution of systems to star categories is given on table 6.4. We notice that the latter doesn't comply with the proposed 5%-30%-30%-30%-5% distribution. This is due to few possible reasons. First the thresholds were developed by using a larger number of systems, so the size of our dataset may not be big enough to act as an appropriate representation of systems. Furthermore, the systems used in our case are open-source systems, which differs to the systems used in [21].

6.3.2 "Dependency Freshness" metric results

The ratings of the systems belonging in our dataset are given in Table 6.5. Furthermore, the metric outputs a report containing the found dependencies and their version sequence number distance to their latest versions.

The distribution of our dataset's system on the star categories is given on table 6.6. The distribution is 19.23% - 34.62% - 19.23% - 19.23% - 7.69%. The difference between this percentages and the proposed 5% - 30% - 30% - 30% - 5% can be explained by the limited size of our dataset and the different origin of the systems consisting our dataset and Cox et. al.'s dataset.

6.3.3 Correlation of ratings

Since all four metrics of our framework have been explained, we examine whether their ratings correlate with each other. Through this, we can determine whether there exist any relationships between the areas our metrics assess.

Furthermore, we check for correlation between the metrics' ratings and characteristics of source code that affect the maintainability of a system. These characteristics are the volume of a software system, the duplication existing on the system's code, the size of its units, as well as their complexity and interfacing, the system's module coupling, and its components' balance and independence.

System	Rating
Clojure	5.5
Elastic Search	5.5
Guacamole-client	5.5
Jetserver	5.5
Java-Websocket	5.5 - *No dependencies* -
Keybox	5.5
MapDB	5.5
OrientDB	5.5
AWS SDK	4.0
Fluent-Http	4.0
HBC	4.0
Libgdx	4.0
Ninja	4.0
Okhttp	4.0
Webbit	4.0
Webmagic	4.0
CAS	3.17
WebGoat	2.5
Oryx	2.17
H2O	1.84
Guice	0.5
Jenkins	0.5
Mahout	0.5
McMMO	0.5
WebScarab	0.5
Internal System 1	-*Confidential*
Internal System 2	-*Confidential*

Table 6.3: "Dependency Vulnerability" ratings.

Star Category	Percentage
5 Stars	29,63%
4 Stars	29,63%
3 Stars	7,41%
2 Stars	7,41%
1 Star	25,93%

Table 6.4: Distribution of systems on "Dependency Vulnerability" ratings.

System	Rating
Clojure	5.5
Keybox	5.5
MapDB	5.5
Fluent-Http	5.5
Libgdx	5.5
Elastic Search	4.22
Ninja	4.19
Okhttp	4.0
OrientDB	3.87
Oryx	3.85
WebScarab	3.76
Mahout	3.76
Jenkins	3.27
Webbit	3.15
WebGoat	3.02
Guacamole-client	2.58
Guice	2.58
CAS	2.45
Webmagic	2.05
H2O	1.97
Jetserver	1.90
HBC	1.64
McMMO	1.4
AWS SDK	1.13
Java-Websocket	- *No dependencies*
Internal System 1	-*Confidential*
Internal System 2	-*Confidential*

Table 6.5: "Dependency Freshness" ratings.

Star Category	Percentage
5 Stars	19.23%
4 Stars	34.62%
3 Stars	19.23%
2 Stars	19.23%
1 Star	7.69%

Table 6.6: Distribution of systems on "Dependency Freshness" ratings.

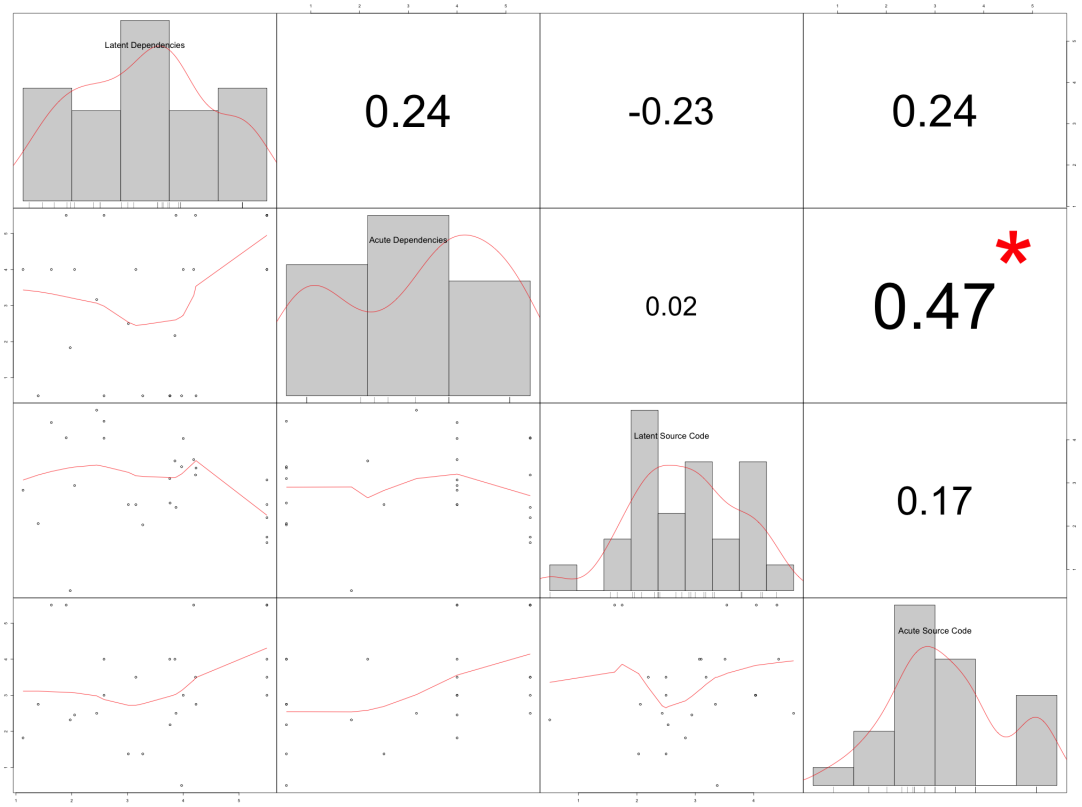


Figure 6.4: Correlation of ratings between metrics.

Our main motivation behind this activity is the discovery of new metrics that can, possibly, be integrated in our framework. Moreover, we could perhaps determine development approaches that could result in more secure software systems.

To acquire the systems' ratings of the maintainability characteristics we used the SAT tool (subsection 5.1.2).

Metrics correlation

Figure 6.4 presents the correlation between the ratings of the four metrics in our framework. We notice a considerate and statistically significant ($p \cong 0.014$) correlation ($Correlation = 0.41$) between the ratings of the "Dependency Vulnerability" and the acute issues metric. All the other metrics don't seem to have a considerate or statistically significant correlation.

Correlation of metrics with maintainability aspects

Figure 6.5 presents the correlation between the ratings of the four metrics in our framework and the maintainability characteristics. There seems to be no considerate correlation between the maintainability characteristics and the ratings of the external dependencies' metrics. Although there is not enough statistical significance to extract conclusive results, this could be explained by the fact that these metrics and the metrics that assess the maintainability characteristics measure different parts of the system, namely the external dependencies of the system and the system's source code, respectively.

The source code metrics and mainly the latent issues metric's rating, seem to have some correlation with specific maintainability characteristics. Particularly, we notice a considerate correlation between the latter and the systems' unit size and unit complexity ($Correlation \cong 0.56$, $p \cong 0.027$ and $p \cong 0.003$), as well as the systems' module coupling ($Correlation \cong 0.62$, $p \cong 0.0007$).

Furthermore, the ratings produced by the acute issues metric present a considerate correlation ($Correlation \cong 0.42$, $p \cong 0.032$) with the ratings of the metric assessing the systems' volume (as we've already seen in subsection5.3.2 [6]). Moreover, negative correlation seems to exist between the rating of the acute issues metric and the unit interfacing of the systems ($Correlation \cong 0.40$, $p \cong 0.044$).

[6] Attention should be paid not to get confused by the fact that these correlations appear to be opposite (-0.377 and 0.42). The first is the correlation between the acute issues metric and the absolute number of the systems' volume in terms of lines of code. The second refers to the correlation of the acute issues metric rating and the rating given to the systems' volume. The smaller the volume of a system, the larger the rating.

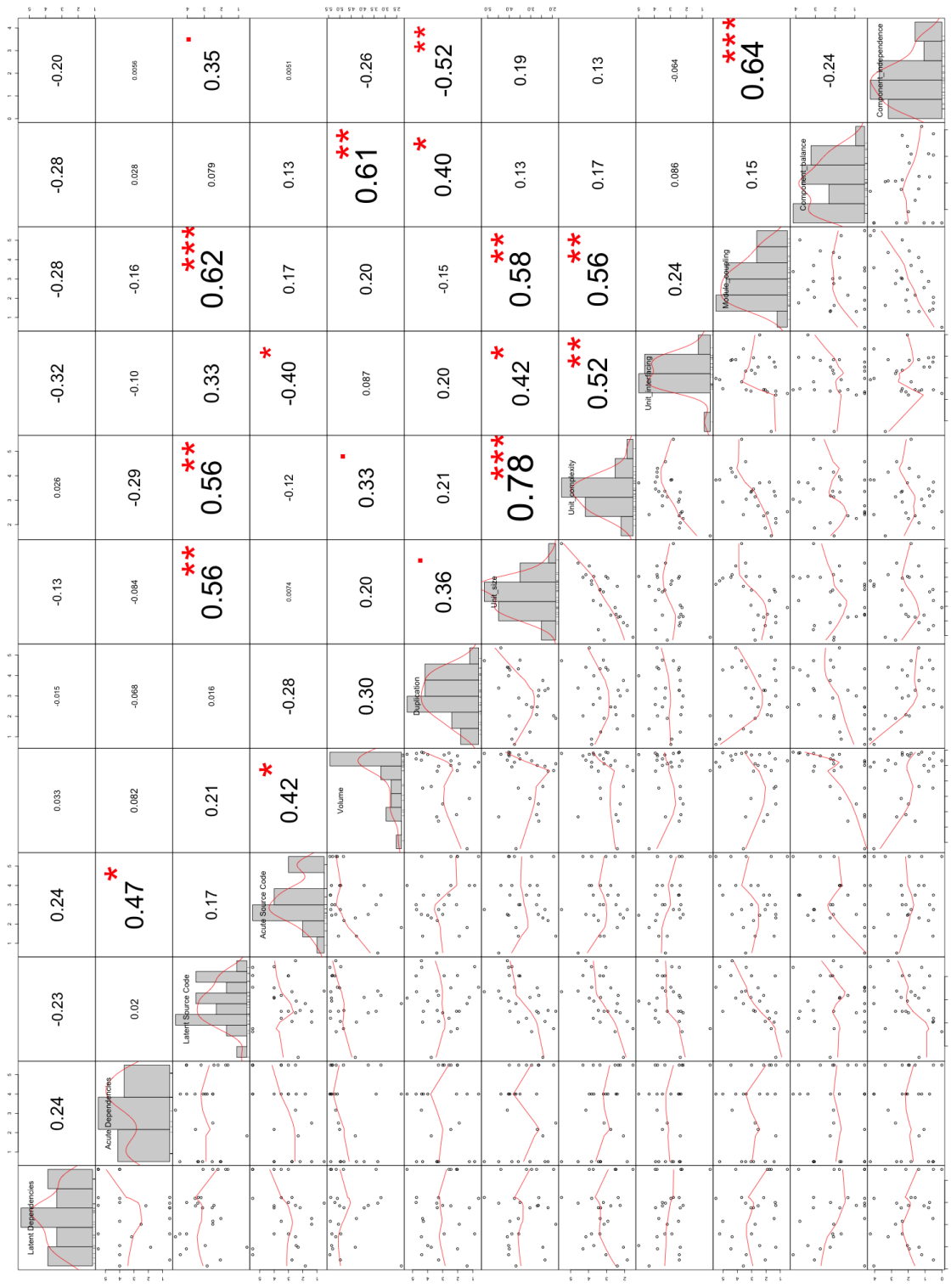


Figure 6.5: Correlation of matrics' ratings with maintainability aspects.

Chapter 7

Validation

This chapter provides details about the design that was used to validate the developed framework of metrics. Moreover, the results of the validation procedure are presented.

7.1 Validation Design

To validate the method and assumptions used to develop the framework, two validation techniques were used.

1. **Interviews.** Semi-structured interviews were conducted to verify the degree of user-acceptance of the proposed model, as well as the metrics conformance to validity criteria such as discriminative power, predictability and consistency.
2. **Longitudinal analysis.** Multiple versions of systems were scanned to identify whether the security ratings of the systems change through time.

The framework is evaluated based on the metrics' validity criteria set by the ISO 1061-1998 standard [13] and for user acceptance determinants proposed by C.K. Riemenschneider [60]. ISO 1061-1998 provides a methodology for establishing quality requirements and identifying, implementing, analyzing and validating process and product quality metrics. The validity criteria defined by the ISO 1061-1998 standard are:

1. **Correlation.** The degree with which the metric rating correlates with

the actual quality factor of the system.

2. **Tracking.** If the quality factor of the system changes through time, the metric value should change. The change should be in the same direction, except if the metric value is inversely related to the quality factor.

3. **Consistency.** If quality factors of systems have distinct relationships then the metric values should also have the same distinct relationships (e.g if $F_1 > F_2 > F_3$ are quality factor values for the systems 1,2 and 3, then their metric values should also be $M_1 > M_2 > M_3$).

4. **Predictability.** A metric should be able to predict a quality factor value with the required accuracy.

5. **Discriminative power.** A metric should be capable of separating a set of high-quality software components from a set of low-quality components.

6. **Reliability.** A metric should be able to pass the validity test over a sufficient number of applications, so that there is confidence that the metric can perform its function consistently.

In addition we will check the framework against significant user-acceptance determinants for methodologies, as elaborated by C.K. Riemenschneider[60]. By examining five theoretical models, of individual intentions to accept information technology tools, the author concluded that the following four determinants were found significant in at least one model:

1. **Usefulness.** A methodology should be regarded as useful by the team members in order to have a successful deployment. For a methodology to be useful, it should increase the team members' productivity and quality of work.

2. **Compatibility.** The methodology should be consistent with the existing needs, values and past experiences of potential adopters.

3. **Subjective Norm / Social Factors.** The intention of team members to use a methodology at workplace is dependent in a large degree in the conceived support that this methodology has from their coworkers and supervisors.

4. **Voluntariness.** Venkatesh and Davis [75] have shown that perceived voluntariness significantly moderates the direct effect of subjective norm on intention to use.

Category	Question
Usefullness	a. Using the framework would improve my productivity. b. Using the framework would enhance the quality of my work. c. The advantages of using the framework outweigh the disadvantages. d. The framework would be useful in my job.
Compatibility	a. The framework is compatible with the way I assess the security of systems.
Subjective Norm	a. People who influence my behavior would think I should use the framework.
Voluntariness	a. Although it might be helpful, using the framework would certainly not be compulsory in my job.

Table 7.1: Questionnaire of User Acceptance determinants.

7.2 Interviews

Four interviews were conducted to verify the degree of user-acceptance of the proposed model, as well as the metrics conformance to validity criteria such as discriminative power, predictability and consistency. By establishing the type of the interviews to semi-structured, we aimed at gaining further insight on the approach of the interviewees to security.

At the beginning of the interview, prior to the presentation of the framework, we chose to focus on the present situation of security assessments and on important properties that a new solution should possess. That was done in order to qualitatively check whether our solution is compatible to the present methodology, potential improvements our solution introduces and important aspects it should conform to. We chose not to present our framework first, since we wanted to avoid bias from the interviewees.

Afterwards, a presentation of the "Metric-based Security Model" was given, describing its goal, its operation, its current state as well as possible future work. That was done as a preface for the next phase, the handing of a questionnaire to the interviewees to quantitatively verify the participant's acceptance of the proposed framework.

The questionnaire used a Likert scale; the questions are presented in Table 7.1.

After the completion of the questionnaire the interviewees were asked

to elaborate on their answers as well as name any missing aspects from the presented framework.

Finally, in the last phase of the interview, the participants were asked to rank five systems based on how they perceived the systems' security level. In order to do so they were presented with material produced by the tools used in the framework. We chose to use unknown to the interviewees systems as to avoid bias.

The last phase was split in two parts. In the first part, output produced by FindBugs was handed to the interviewees, to verify the "Acute" and "Latent Issues" metrics discriminative power. We chose not to, also, verify separately the discriminative power of the external dependencies metrics as similar work was done in [27] and [21].

In the second part, additional material produced by Dependency Check and the Dependency Freshness metric, were handed and the interviewees were asked to re-estimate their judgment. This was done in order to verify the discriminative power of the framework in its entirety.

Four interviews were conducted, of approximate duration of 45 minutes each. The interviewees were all technical consultants, working at SIG. The participants are considered highly experienced software developers, having also performed a number of software security assessments. Subsection 7.2.1 covers in detail the questions that were asked in each phase.

7.2.1 Interview Guide.

As explained in section 7.2, the content phase of the interview was divided in four stages:

Insight into current approach.

1. How do you approach the security assessment of a software system?
2. Which steps could be considered as the bottleneck in your current procedure?
3. Which steps could produce more effective results?
4. What properties do you consider vital in a framework assessing software security?

Introduction of the framework.

In the current phase a presentation of the framework was given, describing its goal, its operation, its current state as well as possible future work.

Questionnaire of acceptance determinants

1. In light of the previous presentation, could you fill in this questionnaire?
2. (Interviewee is asked to elaborate on his answers).
3. Could you name any aspects you think are missing from the framework?

Discriminative Power

1. Given the following material, if you were to rank the five following systems, regarding their security level, what rank would you give?
2. Given additionally the following material (reports from Dependency Check and Dependency Freshness sub-metrics), would you change this ranking, and if yes, how?

End of Interview.

7.2.2 Results.

In this section the results of the interviews will be provided. First the answers of the participants will be given, regarding the current approach followed. Afterwards the answers on the questionnaire of the acceptance determinants, will be presented, followed be the ranking of the systems, as done by the interviewees. A discussion of the results of the validation process, as well as threats to its validity will be provided in Chapter 8.

Insight into current approach answers

Question 1: How do you approach the security assessment of a software system?

Interviewee A: We have a specific product, called "Security Risk Assessment" for that and there is a process that is really simple but there is a number of topics that we typically address there. It depends a little on the exact questions that the customer asks, we may look on everything or sometimes just a specific area. We have a number of areas that we look at, which ranges from technical aspects, such as implementation, source code,

configuration, etc, to architecture, meaning how the system was designed, to process, how it is maintained and developed. Above that there is the governance, the risks on an organization level, that are accepted or not accepted, how the system is managed. So these are typically the areas that we look at a software risk assessment. Our company has a guide for that, and it, fortunately, matches with what we do.

-Regarding source code, how would you approach the issue of security on the source code of a system?

First of all I typically try to understand the architecture of the system, before I actually go to look at the source code, because that helps me understand why certain things were done this way. Sometimes it can be a decision not to secure against a problem which can be a valid decision maybe, because it is an internal application or maybe because it costs too much based on what the harm would be if it was circumvented.

If then we move to the actual source code, I try to see if the architecture that was planned, was actually implemented. We have a security model which is sort of a long list of best practices and it checks if they are applied, that is a very thorough look at the source code, but there is also stuff that is not in there, like it does not check if the project actually implements the designed architecture, or some deliberate design decisions, that the security model may interpret as bad. So the security model is very important. It helps you to look at the source code at a very structured way, so if you apply that you learn a lot.

– As I understood it is not quite often that you search for actual vulnerabilities on the source code, like on a line level?

No, that is right. Well you can do that but that would take forever, on a reasonably sized system. A tool like Checkmarx does try to do that of course but it does have a limited set of vulnerabilities and limited capacity to do that. Well we could spend like two years and review every line and say here is a vulnerability, but that is not very effective. So what we do with the model is to see if the written best practices are applied.

Interviewee B: We look for vulnerabilities originating from relevant to the system threats, which are system-dependable. When we scan the source code, we do not do it randomly; we look in a number of checks that are in our security model, but we also look at how those tests relate in performing those kinds of attacks (that relate with the plausible threats to the system). So basically, we check from a scenario point of view, not a technical point of view.

Interviewee C: It depends on what is important for the client. For example we may look at the process that the developers follow, e.g what kind of secure development methodology do the developers follow, or what coding guidelines they follow. And only a very minor part of our work is to look on the source code. If we had to look on the source code we would do a best effort approach, so we would spend a certain, large, amount of time on code reviews trying to locate issues, after interviewing the developers, trying to understand of the most important transactions through the system, and we would start with those, so we would start from the input, work our way back to a database if its there, and then work to the front again, to basically check whether everything is in place, like escaping, input validation, output encoding. And so, that would be pretty labor intensive, depending also on the quality of the system.

Interviewee D: We typically do manual reviews of the code, based on a checklist of best practices and, on occasions, source code analysis, depending of the requirements of the clients. Additionally, we conduct interviews, trying to locate security problems in the system's architecture and other aspects.

Question 2: Which steps could be considered as the bottleneck in your current procedure?

A: For the source code specifically or for the entire process?

– *Source code.*

The security model itself. Applying that to a reasonably sized system, just takes very long. So, currently we plan about one to two weeks, depending on the system's size and the technology for applying the full model. There exist basically two things that make it take so long. One of them is that the model is quite detailed, all in all there are like 80 to a 100 points that you should check. Even if you take 100,000 lines of code, which is not that big, that is just a lot of work to make sure that you understand each of these things for that system. And the second part is that even if it is a technology that you are quite familiar with, then they are using frameworks that you have never hear of, or old versions, so you just need to figure out how things work with that technology.

B: I would put it quite differently. I think the biggest bottleneck we have is knowledge. With security there is so much knowledge you need to accumulate before you start doing this, that is not good for people to send them for a security assessment in their first year, because it is too much stuff that you have to pay attention to. So another thing is that it is not objective and it is not automated, it depends heavily on who is doing the

assessment. So, it relies too much on the person doing it.

C: In general code inspections are labor intensive, you really need to understand the application well, so before you can even start doing a code inspection you really need to have a really good understanding of its architecture, of the technology stack that is being used, if it is something you are not familiar yourself then you need to investigate as well. So number of bottlenecks: it requires a lot of knowledge, a lot of technology specific knowledge, the code review itself is time-demanding, depending on the size of the system, and how many symptoms you have found, if you find a lot of issues then it is easy, but if you do not find a lot, especially with a large system, then it is harder. Usually also, you do not want to only report the findings but you want to report solutions as well. So, actually all of those can be pretty labor intensive.

D: The source code analysis could be considered as a bottleneck in our current process.

Question 3: Which steps could produce more effective results?

A: What might be problematic, which we have tried to accommodate a bit, is that I might find things that you would not find and you might find things that I would not find, just because of our background, our knowledge maybe the amount of time you put in. So yes, applying the model could be much more effective either by ensuring that everybody does the same thing and maybe having some tools to speed it up.

B: The finding stuff can be more effective. How to make something less dependent to individuals, to standardize the process, to standardize what you look at and automate it, to have some quality checks. Something like your model, a more product-characteristic-based; that is the way we want to go for, because that is objective. Looking at the context will always depend on expert opinion; purely the underlying function though, i.e whether the product is indeed secure or not, that can be automated.

C: It would be nice to have some kind of a repository of code samples in all software technologies. We have some coding guidelines we wrote for customers, but they are pretty mainstream and not technology specific. So a kind of repository technology combined with a countermeasure, that would help a lot, so you immediately knew what you should be doing.

D: The manual review of the source code. Automation could provide better results in terms of repeatability. However, static analysis misses conceptual mistakes that human experts can find.

Question 4: What properties do you consider vital in a framework assessing software security?

A: I think one of the properties you would want from a new model is, on one hand it should of course be correlated to actual security but another thing we are, of course, looking for is that it should be more automated at least, maybe fully.

B: The outcome should be in a way, that someone who owns the system could understand it. The level of risk, acceptable for a system, should be clear.

C: Automation, having a fully automated security model.

D: A new model should provide easy to communicate results, speak in terms of urgency, not by simply listing vulnerabilities. As a result, it should prioritize the findings.

Answers on questionnaire

Table 7.2 presents the answers on the questionnaire of the acceptance determinants (Table 7.1), the third phase of the interview process. The answers of the questions used a likert scale of 5 points. The point with number 1 meant that the interviewee completely disagreed with the question, whereas the point with number 5 meant that the interviewee completely agreed with the question.

In general, the framework is considered useful by the participants (with an average of 4.5 out of 5) with a potential to enhance their quality of work (3.5 out of 5) and increase their productivity (3.5 out of 5). Furthermore, the participants believe that the proposed solution is compatible with the way that they assess the security of systems (3.75 out of 5). Finally, the answers regarding the subjective norm and the voluntariness of use of the framework were close to the median.

After the presentation of the "Metric-based Security Model" the participants were asked to mention if there are, in their opinion, any missing aspects from the framework. Their answers are given below.

Could you name any aspects you think are missing from the "Metric-based Security Model?

A: Some metrics that we do not have yet, in new research areas such as the attack surface of a software system. Furthermore, files with historically,

Question	*A*	*B*	*C*	*D*	**Mean**
1. Using the framework would improve my productivity.	2	5	4	3	**3.5**
2. Using the framework would enhance the quality of my work.	4	2	5	4	**3.75**
3. The advantages of using the framework outweigh the disadvantages.	3	2	4	5	**3.5**
4. The framework would be useful in my job.	4	4	5	5	**4.5**
5. The framework is compatible with the way I assess the security of systems.	4	2	5	4	**3.75**
6. People who influence my behavior would think I should use the framework.	2	2	3	4	**2.75**
7. Although it might be helpful, using the framework would certainly not be compulsory in my job.	5	4	1	3	**3.25**

Table 7.2: Answers of interviewees on questionnaire of acceptance determinants.

many detected vulnerabilities of the system, could be detected by the commit file, or the number of changes or bug trackers. Basically we could look at circumstantial evidence that point to the fact that something is not secure.

B: It is impossible to get it into the model but exploitability, how much could go wrong in practice if I exploit this kind of stuff. Another thing that is also not very practical to put in there is also consistency, whether similar problems are solved in similar ways in the code, because if they do not, then there is more likely chance there is a mistake. But it is also a broad topic. Also, authentication checks, for example authorization checks, you also want them to be implemented in the same way. But in terms of the material that is there I think it is pretty good, but it is not a complete set yet, all of the other stuff though is very difficult to have. Furthermore, error handling could be added, how do people deal with errors in general.

C: I can not think of any missing aspects right now.

D: I tend to say that it is missing the presentation part, a way to communicate the problem.

Discriminative power of metrics

The participants were, additionally, asked to rank five systems based on measurements provided by the tools used in our framework. This was done in two stages.

First, the interviewees were given output provided by FindBugs and "Find Security Bugs" plugin. The target of this stage was to verify the discriminative power of the source code metrics combined, i.e. how accurately they can distinguish systems. The results are presented in Table 7.3. As it can be seen the average ranking of systems by the interviewees is similar to the ranking of the systems as done by the framework. Indeed, in Table 7.5 we see that the Spearman correlation of the interviewees answers to the framework's ranking is 1.

In the second part, additional material produced by the external dependencies metrics, were handed and the interviewees were asked to re-estimate their judgment. This was done in order to verify the discriminative power of the model in its entirety. The results are presented in Table 7.4. Again the interviewees' ranking is similar to the framework's ranking, except for a same average ranking of the first two systems. The Spearman correlation of the interviewees answers to the framework's ranking is 0.974.

The power mean of the metrics' ratings (Equation 4.1) was used to get the framework's combined ranking of the systems. On the first part, the ratings produced by the source code metrics where aggregated, whereas in the second part the ratings produced by all the metrics of the framework, were used.

System	**Rank of Systems** *(By framework)*	*A*	*B*	*C*	*D*	**Average**
HBC	1	1	2	1	1	1.25
Ninja	2	2	1	2	2	1.75
CAS	3	5	3	3	3	3.5
WebScarab	4	3	4	4	5	4
H2O	5	4	5	5	4	4.5

Table 7.3: Answers of interviewees on ranking of systems based on source code static analysis tools output.

System	**Rank of Systems** *(By framework)*	*A*	*B*	*C*	*D*	**Average**
Ninja	1	2	1	2	1	1.5
HBC	2	1	2	1	2	1.5
CAS	3	5	3	3	3	3.5
WebScarab	4	3	4	4	5	4
H2O	5	4	5	5	4	4.5

Table 7.4: Answers of interviewees on ranking of systems based on output of all the tools used in the framework.

Part	**Spearman Correlation**
Output from tools used in source code metrics	1
Output from all the tools used by the framework	0.974

Table 7.5: Spearman correlation of interviewees ranking of systems to framework's rankings.

7.3 Longitudinal Analysis.

The second part of the validation procedure is consisted of a longitudinal analysis of the source code metrics, on five systems of the dataset. The target of the analysis is to identify whether the metrics' ratings change through time. Through this we aim to verify whether the metrics conform with the "Tracking" validity criterion of ISO 1061-1998. We decided not to include the external dependencies metrics in the analysis, since similar work has been done in [21] and [27].

The tracking criterion states that if the quality factor of the system changes through time, the metric value should change. The change should be in the same direction, except if the metric value is inversely related to the quality factor.

In the analysis we used 40 snapshots of 5 systems, 8 per system. The systems, their versions and the release dates of the versions can be seen in Table D.1 of Appendix D.

7.3.1 Results

The ratings for both the "acute" and "latent" issues metrics, for each system's version, can be seen in Table D.2 of Appendix D. Furthermore, Figures 7.1a and 7.1b show the change of the "acute" and "latent" issues metrics respectively, per system's version.

We notice that the ratings can vary per version, especially in regard to the latent issues metric. This change is caused by the addition or removal of code containing potential vulnerabilities. Table 7.6 shows the variance of ratings for each system. For the latent issues the system with the highest variance is Webbit (0.3310) whereas Ninja has the lowest variance between its ratings for each version (0.1182). Ninja and Fluent-http show no variance between their acute issues ratings. Webbit, on the other hand, has also the highest variance of its acute issues ratings (0.0714).

In general, the ratings produced by the latent issues metric show higher variance than those produced by the acute issues metric. Table 7.7 presents the median and mean of the variances for each metric. The median and mean for the latent issues ratings are 0.1369 and 0.1681, whereas for the acute issues they are 0.0078 and 0.0266 respectively.

As a next step we examined whether the degree of change of ratings per version of a system is dependent on the difference of volume between the two versions. Figure 7.2 and Table D.3 (located in Appendix D) show the size difference, in terms of lines of code, between subsequent versions of the systems. To identify whether a larger change of volume between two versions could result in a larger change of ratings, we checked for the Pearson correlation between the absolute value of the size difference between the two versions and the absolute value of the difference of their ratings. The results can be seen in Table 7.8. Although the p-values are quite high to allow for statistically significant results, we notice that there is very low to no correlation (0.1101 for the latent issues and 0.0419 for the acute issues).

System	Variance of Latent Issues Rating	Variance of Acute Issues Rating
Guacamole	0.1505	0.0078
Webbit	0.3310	0.0714
Ninja	0.1182	0.0000
OkHttp	0.1369	0.0536
Fluent-Http	0.1038	0.0000

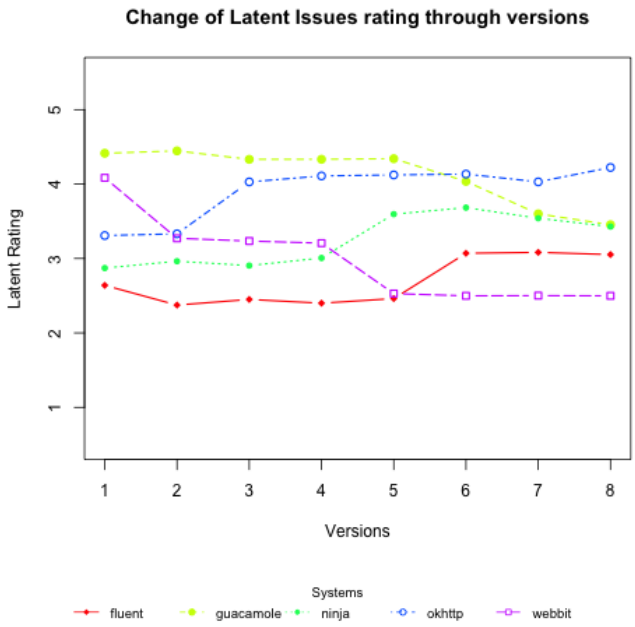
Table 7.6: Variance of rating changes.

Rating	Median	Mean
Latent Issues	0.1369	0.1681
Acute Issues	0.0078	0.0266

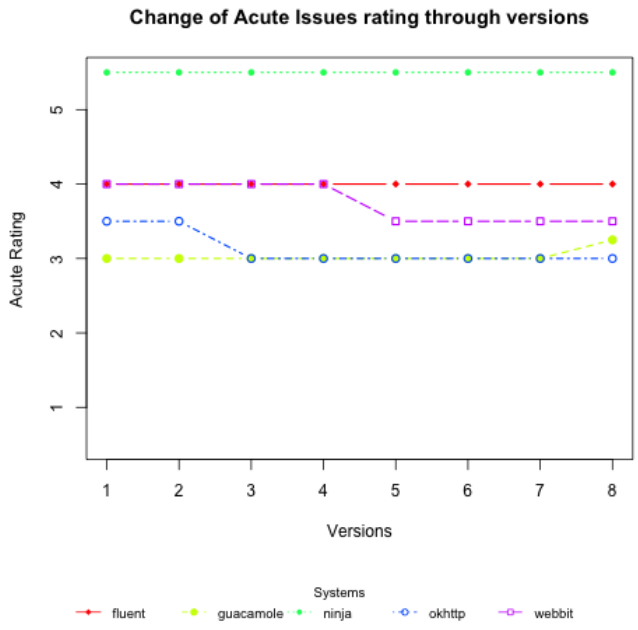
Table 7.7: Median and mean of variance of rating changes.

Rating	Pearson Correlation with size Difference	p-value
Latent Issues	0.1101	0.4986
Acute Issues	0.0419	0.7974

Table 7.8: Correlation between absolute size difference and absolute rating change, between versions of the systems.



(a) Latent Issues Ratings of systems.



(b) Acute Issues ratings of systems.

Figure 7.1: Ratings of systems.

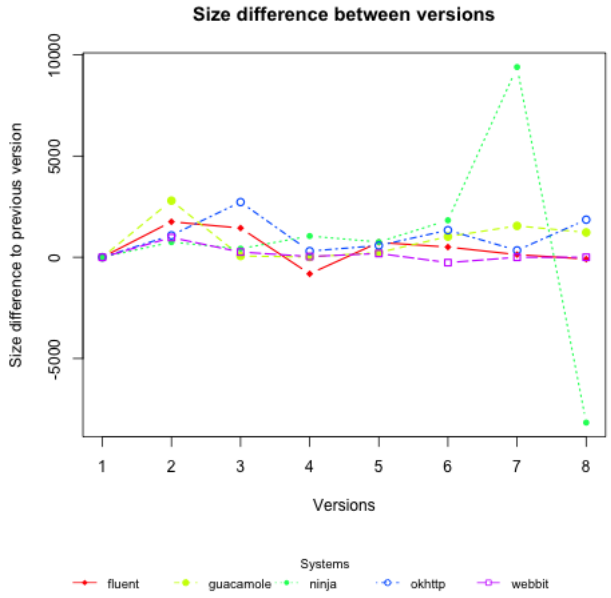


Figure 7.2: Size difference between subsequent versions of systems.

Chapter 8

Discussion

In the current chapter we provide our interpretation of the results produced by the application of the framework in the dataset of open source systems we used, as the former were presented in sections 5.3 and 6.3. Furthermore, we comment on the results of the validation procedure (Chapter 7), and finally, we state the factors that act as threats to the validity of our study.

8.1 Discussion on result sections

8.1.1 Discussion on results of source code metrics.

On the results section of the "Source code metrics" chapter (Section 5.3) we provide an example of how a system is rated by both the "Acute" and "Latent Issues" metrics. Furthermore, we present the distribution of the systems of the dataset in the star categories as well as the effect of the rating methods used, in the correlation of the ratings and the systems' size.

As it can be seen, the rating methods used are efficient both in regards to the distribution of the systems in the star categories, where the actual distributions were close to the targeted ones, as well as on providing little correlation between the rating of a system and its size (-0.053 for the "Latent Issues" and -0.377 for the "Acute Issues"). The reason that we consider important that little correlation should exist can be seen in Section 5.2.2. There, by conducting a statistical analysis of the systems in our dataset we proved that the correlation of the number of potential vulnerabilities reported by FindBugs and the systems' size is 0.966 for the "Latent Issues" and 0.415 for the "Acute Issues". Therefore, in the case of the "Latent

Issues" metric we decided to take into account the size of the system in the rating method, in order to avoid larger systems having a lower rating than smaller systems.

On the contrary, regarding the "Acute Issues" metric we chose not to normalize by system size. This decision was taken for two reasons. First the correlation we found in the statistical analysis of the section 5.2.2, as well as the correlation reported by Mitropoulos [53] of the "Security High" category (where our categories originate from) is not as high as the correlation of the reported vulnerabilities of the "Latent Issues" metric with the systems' size. Furthermore, we believe that, due to the severity of the vulnerabilities belonging in the "Acute Issues" metric, a rating method that would assess the absolute number of vulnerabilities found would be more appropriate.

8.1.2 Results of third-party dependencies metrics.

In Section 6.3 we present the results from the application of the external dependencies metrics on the dataset. At the beginning the ratings and additional output produced by each metric are provided. Afterwards, we examine whether there is any correlation between the framework's ratings. Finally, we check whether any correlation exists between our metrics ratings and ratings from metrics that assess maintainability properties of the systems' source code.

On both metrics of the external dependencies we used thresholds that were found through research that was already conducted prior to ours. In the case of the "Dependency Vulnerability" metric the thresholds used were identified in the context of the research conducted by Cadariu [21], whereas as already explained, the "Dependency Freshness" metric was created by Cox [27]. We chose to use the authors' thresholds rather than identifying our own as the number of system that were used to define the thresholds were considerably more, which could result in more accurate thresholds. Furthermore, in the case of the "Dependency Vulnerability" metric, we were enabled to use these thresholds since in both our research and in Cadariu's research the same tool, "Dependency Check", was used to detect the vulnerabilities in the third party dependencies of the systems.

We notice, though, that for both metrics the distribution of our dataset's systems, in the star categories, did not fully comply with the targeted distributions. As we explained, we believe that a reason for this is that, due to its limited size, our dataset was not representative enough.

Furthermore though, in the case of the "Dependency vulnerability" met-

ric we should also consider that there are differences between both the version of the "Dependency Check" that was used in ours and Cadariu's research, the location from where the tool was acquiring the third party dependencies used by a system (in the case of Cadariu's research the .pom file was accessed whereas in our research the tool was using the .jar files of the dependencies), and the further classification of the reported vulnerabilities to confidence levels that was conducted in our research (and through which only the high and highest confidence vulnerabilities were taken into account in the rating of the systems).

In the same section we also check for correlation between the ratings of the framework's metrics. The results are given in Figure 6.4. We notice that the ratings produced by the "Acute Issues" metric and the "Dependency Vulnerability" metric have a correlation of approximately 0.47. That could mean that in systems with severe security violations in their source code there is also not proper control for known vulnerabilities in their external dependencies. On the other hand, we notice that there is no significant correlation between the other ratings that were produced by the metrics. We believe that this can indicate that, indeed, the characteristics that each metric assesses are separate and the performance of a system in one metric doesn't affect its performance in another. That fact, in turn, indicates the usefulness of our framework, since as already noted in related literature [45] no metric by its own can produce conclusive results about the security level of a software, so a framework that assesses a system from multiple sides is needed.

We additionally checked whether certain maintainability characteristics can indicate the security status of a system. Our motivation behind this experiment is the addition of more metrics and as a result of more sources of information, which can provide a more complete view to the stakeholders of a system. The results are presented in Figure 6.5. As we notice the metrics of the external dependencies show no correlation with the maintainability characteristics of the systems. This can be explained by the fact that the two sides assess different parts of a system, namely its external dependencies and its source code respectively. On the other hand, the source code metrics and mainly the "Latent Issues" metric's rating, seem to have some correlation with specific maintainability characteristics. Particularly, we notice a considerate correlation between the latter and the systems' unit size and unit complexity ($Correlation \cong 0.56$, $p \cong 0.027$ and $p \cong 0.003$), as well as the systems' module coupling ($Correlation \cong 0.62$, $p \cong 0.0007$). That can indicate that as a system becomes harder to maintain it becomes easier to introduce latent vulnerabilities on it.

Furthermore, the ratings produced by the acute issues metric present

a considerate correlation ($Correlation \cong 0.42$, $p \cong 0.032$) with the ratings of the metric assessing a system's volume. That was expected and this issue was already addressed in Subsection 8.1.1. Moreover, negative correlation seems to exist between the rating of the acute issues metric and the unit interfacing of the systems ($Correlation \cong 0.40$, $p \cong 0.044$). The later indicates that units with more parameters defined in their signature or declaration are more possible to have an acute issue vulnerability.

8.1.3 Results of validation.

Chapter 7 presents the design and results of the validation of the framework. The aim of this chapter, specifically, is to assess the framework's conformance to user acceptance criteria, as well as check the metric against the validity criteria specified by ISO-1061-1998 [13].

Two validation techniques were used. First, four semi-structured interviews were conducted. The interviewees were all, experienced developers, as well as consultants internal to the company where the research was conducted, with wide experience in software security assessments. The second part of the validation procedure was consisted of a longitudinal analysis of the source code metrics, on five systems of the dataset.

Interviews.

The interviews were split into four parts. At the beginning questions regarding the participants current approach of security assessments were asked. The purpose of these questions was to, qualitatively, assess whether our proposed framework is compatible with the way the experts assess the security of a system. Namely, whether it addresses the disadvantages that their approach currently has and whether it adds features that they consider vital.

The first question asked the participants on what is their current approach towards assessing the security of source code. The participants are conducting, if needed, manual code reviews based on a set of guidelines that they use. These guidelines, mainly assess whether best practices are generally applied but they do not look on a source code level to identify whether actual vulnerabilities are present. Their approach possesses both advantages and disadvantages. Namely, the interviewees include the architecture and design of the system in the aspects that they assess as well as the context within the system is used. Furthermore, they overall check more aspects than our framework does. For both these reasons their approach can po-

tentially provide a more holistic view of the security status of a software system.

However, on the other hand, this approach does not (with the exception of the occasional use of a static analysis tool) look in depth whether specific vulnerabilities are present in an application, which could result in a potential loss of vital information. Moreover, these manual reviews are time consuming, dependent on the expert's opinion and knowledge and their results are hard to reproduce.

These disadvantages were the subject of the next two questions. More specifically in the second question the participants were asked which step was the bottleneck (in terms of time) in their current approach. Our rational behind this question was that we consider, as an advantage of our framework the fact that it is automated which could result in a significant decrease of the time needed to conduct an assessment. Indeed, the participants verified that these manual code reviews were highly time-consuming, both due to the number of separate issues that had to be checked as well as due to the time needed for the person that conducts the assessment to become familiar with the system and the technologies used.

The third question asked the participants which steps of their process could produce more effective results. This question differs from the previous one since it does not focus on the time needed to complete the assessment but rather in the end quality of the results produced. A number of factors has been given. The participants, more specifically, believe that the end quality of the assessment is dependent on the level of knowledge and experience of the assessor. Furthermore, they think that if their process was supported by automated tools, the discovery of results could become more efficient. Moreover, they note that further standardization of the process could only be achieved by further automating it. Another advantage of introducing automation in their process, they mentioned, would be the improvement in terms of repeatability of the results. Their answers prove, hence, that our framework can prove useful for improving their current methodology.

The fourth question of the interview focused on aspects that the participants consider vital in a new methodology that would assesses the security level of a software. The participants believe that a new methodology should be as automated as possible and its results should, evidently, correlate to the actual security of the system. Furthermore, the interviewees believe that the results should be easy to communicate, support a prioritization of findings and set an acceptable level of risk for an application. We consider that our methodology satisfies these demands. The framework was designed to use automated metrics in order to reduce the time needed to complete an

assessment and enable easy repeatability. Furthermore, by using the current rating method, we consider that the results of the assessments are easy to communicate and by having a separate rating for each metric we believe that the findings can be prioritized.

In the second part of the interviews a presentation of our framework was given. The latter was followed by a completion, from the participants, of a questionnaire assessing the framework's conformance to user acceptance criteria. The participants answered, in total, seven questions regarding the framework's usefulness, compatibility to their current approach, subjective norm and voluntariness of use. It should also be noted that the interviewees were asked to rate the framework based on its condition at the time that the interviews were conducted, rather than its perceived condition when the framework would be completed. This fact, as the participants mentioned, resulted in a lower rating.

In general though, our solution is considered useful by the participants (with an average of 4.5 out of 5) with a potential to enhance their quality of work (3.5 out of 5) and increase their productivity (3.5 out of 5). Furthermore, the participants believe that the proposed solution is compatible with the way that they assess the security of systems (3.75 out of 5). Finally, the answers regarding the subjective norm and the voluntariness of use of the framework were close to the median, although the participants noted that the current condition of the model affected their reply.

We consider that the participants' answers on the questionnaire confirm our beliefs that the framework is useful in order to assess the security status of an application. On another note though, possibly, the advantages of using the model should have been communicated better. The participants, also, consider as a disadvantage of the framework the number of false positives that the tools produce, a point that will be described further in the next section.

The participants were, afterwards, asked to name any missing aspects from the framework. Their answers provided a number of new metrics that could be added in the framework. Furthermore, it became understood that more work could have been done in the presentation of the findings since an interviewee thought that this part could be improved.

In the final part of the interviews the participants were asked to rank five systems based on measurements provided by the tools used in our framework. This was done in two stages. First, the interviewees were given output provided by FindBugs and "Find Security Bugs" plugin. The target of this stage was to verify the discriminative power of the source code metrics com-

bined, i.e. how accurately they can distinguish systems.

In the second part, additional material produced by the external dependencies metrics, were handed and the interviewees were asked to re-estimate their judgment. This was done in order to verify the discriminative power of the model in its entirety.

From the results we notice that, in general, the interviewees judgment was close to our framework's especially from distinguishing the highest ranking systems from the lowest ones. We contribute some variations from the exact order of the systems in the following factors. First, certain systems had only slight differences in their findings, which could make it harder for an individual to rank them. Furthermore, we noticed that the interviewees took into account the context within these systems are meant to be used, something that is not yet addressed by the framework. Finally, the participants mentioned that there was provided a large amount of information which made its processing harder. In fact, we consider the final comment as another issue that an automated solution can address.

Longitudinal Analysis

The second part of the validation procedure was consisted of a longitudinal analysis of the source code metrics, on five systems of the dataset. The target of the analysis was to identify whether the metrics' ratings change through time.

Through the analysis we verified that, indeed, the ratings of the metrics can show variation between consecutive versions. We noticed, though, that this variation is significantly larger for the "Latent Issues" metric's ratings than for the "Acute Issues". We consider that this difference is caused by the small number of acute issues per system for our dataset, the increased difficulty that there is (in comparison with the latent issues) in introducing new acute vulnerabilities in a system and in the limited size of our dataset which made the effect of outliers larger during the calculation of the thresholds in our rating method. The latter resulted in smaller rating differences between different distinct values of acute issues.

We also identified that the difference in size between versions doesn't affect the difference in their ratings. Although a person could support that the bigger the changes are, the larger the chance for an issue to be introduced in a system, on the other hand we consider that new vulnerabilities can be created to a system even from a small change to its source code. Moreover, during our analysis we noticed that the more mature a system is, it presents

smaller rating changes. However, since the purpose of our analysis was different, further research is needed in order to produce a conclusive result on that field.

8.2 Threats to Validity

During our research we encountered a number of potential threats to the validity of our work. Those are summarized below.

8.2.1 Accuracy of static analysis tools.

The accuracy of our results depends on the accuracy of the static analysis tools we use. One of the major disadvantages of the latter is the number of false positives they produce. As already stated in section 2.1.3 there is a considerate amount of false positives produced by the tools, which varies from 20% to even 100%. Although some minor work has been done, within our research, to limit the effect of false positives on the validity of our framework (such as the filters designed and the rating of only the high and highest confidence level vulnerabilities in the "Dependency Vulnerability" metric), it should be noted that it is not a goal of our research to evaluate the accuracy of the static analysis tools used and hence we didn't distinguish between false and true positives before producing a rating for the systems of our dataset. As it can be understood then, the accuracy of the final ratings of the systems and of their reported vulnerabilities can be significantly affected by this fact.

We should also note that the tool we used for the source code metrics, i.e FindBugs, is not a specialized tool in identifying security vulnerabilities. It could be suggested, hence, that a security specific static analysis tool could produce findings of higher quality. The reasons though that we chose this specific tool are explained in section 5.1.1. These include the fact that it is an open source tool, opposite to most security static analysis tools which are commercial, it is faster in analyzing a system and it has been used extensively in previous research.

Another disadvantage of automated source code scanning, in general, is its inability to take into account, during the analysis of a system, the context within the latter is used. As a result, a static analysis tool can report vulnerabilities which are not possible to be exploited or which have limited or no impact in the security of the system and its information. Furthermore, we should also note that automated tools of this type do not assess security

flaws originating from flawed architectural design of a system, as well as implicit mistakes from misconceptions of how a system should operate.

Finally, although, static analysis tools can detect vulnerabilities on existing source code, they do not have the ability, and hence neither does our framework, to detect complete absence of security controls or other security specific functions of a system.

8.2.2 Method design.

We consider as a constraining factor to the validity of our research the limited size of our dataset. As already mentioned in our thesis, the latter causes implications in the accuracy of the thresholds produced, where 27 systems could be considered as a limited sample. Moreover, as we have seen in the statistical analysis conducted in section 5.2.2, we were able to identify only 24 out of the 46 distinct acute vulnerabilities and 15 out of the 43 latent vulnerabilities, within our dataset.

The generalizability of our research is also limited by the type of the applications our framework can rate. More specifically, as explained in section 4.3 all the systems in our dataset are Java systems and Maven compilable. This fact is caused due to the implementation details of our metrics and tools used, since FindBugs and "Dependency Check" can only scan Java systems, whereas "Dependency Freshness" reads the .pom files needed in maven-compilable systems.

8.2.3 Validation design.

The limited number of participants, in the interviews conducted during the validation of the framework, could also introduce variations of the results produced from the actual acceptance of our solution. Furthermore, the fact that the interviewees were all employed within the company that the research was conducted could introduce bias in their answers. Finally, more types of stakeholders of an application could have been interviewed such as full-time developers and managers, in order to evaluate different aspects of our framework and each category's distinct acceptance to our solution.

In defence of our validation design we should note that its target was to extract qualitative information regarding the framework's acceptance and validity. In that context we consider that the number of the participants is adequate. Furthermore, by interviewing experienced consultants we were able to harvest information addressing multiple sides of interest regarding a

software system and its level of security.

Chapter 9

Conclusion

In the given study we demonstrated how a framework of metrics can be created in order to assist a system's stakeholders into acquiring a quantitative basis of its security status.

The main research question of this study, as presented in section 1.1, is:

How can software security be expressed by means of automatically-obtained vulnerability metrics?

In order to answer the main research question the following sub-questions have been determined:

1. *How to derive metrics from a system's software vulnerabilities?*
   (a) *Can we assess a system's security by deriving metrics that rely on the severity of software vulnerabilities?*
   (b) *Can we derive metrics from static analysis tools' automated measurements?*
2. *How to build a framework of metrics to assess software security?*

We followed a Goal-Question-Metric approach to identify a set of metrics that enable us to assess the security level of a software (Section 3.2). As can be seen in Table 3.1, the metrics that consist the framework, address different aspects of a system's security.

The first two metrics, "Acute" and "Latent Issues", detect vulnerabilities within the source code of the system and provide a rating based on the number of vulnerabilities detected.

The "Acute Issues" metric detects high severity violations of secure coding patterns. These violations require no or minimal knowledge of the application's internal structure and are related with issues such as injection vulnerabilities, unvalidated redirects, sensitive data exposure, etc. The "Latent Issues" metric detects and provides a rating for medium severity violations detected, such as vulnerabilities for which another program should be written to incorporate references to mutable objects, non final fields, etc.

The other two metrics detect issues located in third-party dependencies of the system. More specifically, the "Dependency Vulnerability" metric detects vulnerabilities located on a system's dependencies and returns a rating based on the number of vulnerabilities found and their severity. Finally, the "Dependency Freshness" metric, detects how recently the third party dependencies of the system were updated.

The first three metrics were developed within this research. The "Dependency Freshness" metric, developed by Cox [27], was integrated to the metric-based Security model, to provide an as more holistic view, of the security status of the third-party components, as possible.

In order to identify vulnerabilities on the software system we used static analysis tools. FindBugs and its security-oriented plugin "Find Security Bugs" were used to locate vulnerabilities on the source code of the system, whereas Dependency Check was used to locate external dependencies with known vulnerabilities. Furthermore, SAT (Section 5.1.2) was used to provide information regarding the size of the software system as well as ratings of maintainability characteristics of the systems in our dataset. The reasons we chose FindBugs are stated in Section 5.1.

The rating generated by each metric shows the status of the system, in regard to the security characteristic analyzed by the metric. In order to define the thresholds of each rating category we used a dataset of 27 systems of which 25 were open-source and 2 were internal systems of the organization within this study was conducted. The rating methods used by each metric are explained in Sections 5.2.3, 5.2.4, 6.1.2 and 6.2.

3. *Is the proposed framework useful for assessing software security?*

4. *Do the proposed metrics accurately express software security?*

The validation procedure of our framework was carried in two parts. In the first part we conducted four interviews to identify the framework's conformance to acceptance criteria for new methodologies as identified by Riemenschneider [60] as well as to evaluate the framework's metrics against validity criteria set by the ISO 1061-1998 standard [13]. More specifically

during the interviews we checked whether the metrics conformed with the "consistency" and "discriminative power" validity criteria.

In the second part of the validation procedure we conducted a longitudinal analysis on five systems of our dataset. The goal of the analysis was to identify whether the source code metrics conformed with the "tracking" and "reliability" validity criteria of ISO 1061-1998. Eight snapshots of each system were selected and their ratings' variance was measured.

Through the creation of such a framework we aimed to prove that multiple gains can be achieved, such as a decrease in time and human resources needed to perform an analysis, an increase in the quality and quantity of the results, as well as a valid representation of the security level of a system.

Our claims are supported through the findings of the Validation chapter (Chapter 7), where the usefulness and validity of our framework were evaluated. Hence, although we acknowledge that an automated method is not yet able to perfectly identify all security risks present in a system, it is our belief that further research towards this direction could lead to more secure systems, based on the enabling of system stakeholders in making more informed decisions.

Concluding this document, we plan to address the contribution this thesis brings to the already existing work conducted in the relevant fields. Furthermore, we will present possible directions towards the continuation of our research.

9.1 Contributions

Our motivation behind conducting the present research lies on the fact that, although several security metrics already exist such as the ones of Cadariu [21], Manadhata [45], Cox [27] and Wang [80], relevant research is still on an initial stage [56][9], mainly because of the difficulty in defining meaningful and efficient metrics.

Furthermore, as Manadhata et al. [45] noted, no single quality metric or measurement is able to holistically assess the security status of a system. Due to this reason, the author states, a framework is needed to combine multiple security measurements.

Through our research we tried to make a contribution on both issues. In that direction, we created three metrics that interpret automated measurements and enable stakeholders, based on quantitative data, to make

informed decisions regarding the security status of their applications.

Moreover we created a framework of metrics, with the additional integration of the work of Cox [27]. Our proposed framework aims in providing a multi-sided view of the security level of an application. The significance of a framework to integrate and correlate multiple security metrics has been mentioned in a number of articles (e.g [45], [74]). At the same time, though, a lack of such frameworks ([45],[33]) was also apparent, a limitation, towards the resolve of which, we believe that we contributed.

9.2 Future work

We distinguish two main directions towards the continuation of our research. More specifically, we believe that progress can be made both by introducing improvements in the already developed metrics, as well as by adding new metrics in our framework.

9.2.1 Current metrics

Regarding the current metrics, we consider that reducing the effect of false positives could significantly improve the accuracy of our framework. Notable work, such as the one already conducted by Kim ([40],[41]) and Heckman [30] could be adapted and integrated in our solution.

Another means for improvement could be the acquisition of more measurements through the concurrent use of additional static analysis tools. The latter could benefit our model both by limiting the number of false positives through cross-checks of measurements between tools as well as by producing a larger number of reported vulnerabilities and hence more effective results.

Further research should also be conducted in order to generalize our framework to be able to assess systems developed in more programming languages than only Java. Moreover, additional validation of the metrics that consist our framework, and of the latter in its entirety, should take place. The validation should be conducted with external to the company, where the research was conducted, participants. Through this we aim to verify the generalizability of our findings, in regard to the framework's validity and user acceptance.

Extending the number of systems that we use as a dataset is also considered as a future step. The latter will increase the accuracy of the ratings

produced, since we will have more datapoints to define the thresholds of the star categories. Moreover, improvements should be made in the presentation of the findings, as was suggested in the validation of the framework.

9.2.2 New metrics

Another direction of future work could be the addition of new metrics. Those could belong either in the four areas that our metrics currently assess or in new ones.

An implementation, for example, of the attack surface metric, originally proposed by Manadhata [45], could be integrated in the latent issues of the source code. Furthermore, by using the taxonomy of the "Seven Pernicious Kingdoms" [73], as described in section 2.1.2, to group the vulnerabilities both in the latent and acute issues of the source code, we could introduce specialized metrics that would assess the status of the system in regards to these seven categories (we consider that the environment category would be covered in other areas of the framework).

The framework, though, could also be extended by creating methods to assess areas that are currently not covered. In the validation chapter (Chapter 7), we mentioned the significance of the design and architecture of a system to its security status. There we supported, based on responses from the interviewees, that fundamental mistakes on the security of an application can be caused by mistakes introduced in the design phase of a system. The creation, hence, of a metric to validate the design and architecture of a system could be of high value.

Bibliography

[1] Apache maven website (https://maven.apache.org).

[2] Cvss website. https://www.first.org/cvss.

[3] Dependency check website.

[4] Find security bugs website (http://h3xstream.github.io/find-sec-bugs/).

[5] Findbugs website (http://findbugs.sourceforge.net).

[6] Pmd website https://pmd.github.io.

[7] Sonarqube's website http://www.sonarqube.org.

[8] Yasca website http://www.scovetta.com/yasca.html.

[9] Omar H Alhazmi, Yashwant K Malaiya, and Indrajit Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228, 2007.

[10] Tiago L Alves, José Pedro Correia, and Joost Visser. Benchmark-based aggregation of metrics to ratings. In *Software Measurement, 2011 Joint Conference of the 21st Int'l Workshop on and 6th Int'l Conference on Software Process and Product Measurement (IWSM-MENSURA)*, pages 20–29. IEEE, 2011.

[11] Tiago L Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[12] Nuno Antunes and Marco Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on*, pages 301–306. IEEE, 2009.

[13] IEEE Standards Association et al. Ieee standard for a software quality metrics methodology. *IEEE Std*, pages 1061–1998, 1998.

[14] Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252. ACM, 2010.

[15] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[16] Dejan Baca, Bengt Carlsson, and Lars Lundberg. Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 79–88. ACM, 2008.

[17] Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg. Static code analysis to detect software security vulnerabilities-does experience matter? In *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, pages 804–810. IEEE, 2009.

[18] Victor R Basili. Software modeling and measurement: the goal/question/metric paradigm. 1992.

[19] Matt Bishop et al. A taxonomy of unix system and network vulnerabilities. Technical report, Technical Report CSE-95-10, Department of Computer Science, University of California at Davis, 1995.

[20] Peter Burris and Chris King. A few good security metrics. *METAGroup, Inc. audio*, 11, 2000.

[21] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. Tracking known security vulnerabilities in proprietary software systems. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 516–519. IEEE, 2015.

[22] Luiz Fernando Capretz, Miriam AM Capretz, and Dahai Li. Component-based software development. In *Industrial Electronics Society, 2001. IECON'01. The 27th Annual Conference of the IEEE*, volume 3, pages 1834–1837. IEEE, 2001.

[23] P Charles and Shari Lawrence Pfleeger. *Analyzing Computer Security: A Threat/vulnerability/countermeasure Approach*. Prentice Hall, 2012.

[24] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

[25] No CNSSI. 4009,". *National Information Assurance (IA) Glossary*, 26, 2010.

[26] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 673–674. ACM, 2006.

[27] Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. 2014.

[28] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *software, IEEE*, 19(1):42–51, 2002.

[29] David P Gilliam, John D Powell, John C Kelly, and Matt Bishop. Reducing software security risk through an integrated approach. In *Software Engineering Workshop, 2001. Proceedings. 26th Annual NASA Goddard*, pages 36–42. IEEE, 2001.

[30] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 41–50. ACM, 2008.

[31] Sarah Heckman and Laurie Williams. A model building process for identifying actionable static analysis alerts. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 161–170. IEEE, 2009.

[32] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[33] Thomas Heyman, Riccardo Scandariato, Christophe Huygens, and Wouter Joosen. Using security patterns to combine security metrics. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 1156–1163. IEEE, 2008.

[34] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[35] Michael Howard, David LeBlanc, and John Viega. *The 19 Deadly Sins of Software Security.* McGraw-Hill/Osborne California, 2005.

[36] Michael Howard, David LeBlanc, and John Viega. *The 19 Deadly Sins of Software Security.* McGraw-Hill/Osborne California, 2005.

[37] Michael Howard, David LeBlanc, and John Viega. 24 deadly sins of software security: Programming flaws and how to fix them. 2009.

[38] SO ISO. Iec 25010: 2011 (2011). *Systems and software engineering–Systems and software Quality Requirements and Evaluation (SQuaRE)–System and software quality models.*

[39] George Jelen. Sse-cmm security metrics. In *NIST and CSSPAB Workshop*, 2000.

[40] Sunghun Kim and Michael D Ernst. Prioritizing warning categories by analyzing software history. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 27. IEEE Computer Society, 2007.

[41] Sunghun Kim and Michael D Ernst. Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54. ACM, 2007.

[42] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Static Analysis*, pages 295–315. Springer, 2003.

[43] Carl E Landwehr, Alan R Bull, John P McDermott, and William S Choi. A taxonomy of computer program security flaws, with examples. Technical report, DTIC Document, 1993.

[44] Steve Lipner. The trustworthy computing security development lifecycle. In *Computer Security Applications Conference, 2004. 20th Annual*, pages 2–13. IEEE, 2004.

[45] Pratyusa K Manadhata and Jeannette M Wing. An attack surface metric. *Software Engineering, IEEE Transactions on*, 37(3):371–386, 2011.

[46] D McCallam. The case against numerical measures of information assurance. In *proceedings of the Workshop on Information-Security-System Rating and Ranking held in Williamsburg, VA*, 2001.

[47] Gary McGraw. Testing for security during development: Why we should scrap penetrate-and-patch. In *Computer Assurance, 1997.*

COMPASS'97. Are We Making Progress Towards Computer Assurance? Proceedings of the 12th Annual Conference on, pages 117–119. IEEE, 1997.

[48] John McHugh. Quantitative measures of assurance: Prophecy, process or pipedream. In *Workshop on Information Security System Scoring and Ranking (WISSSR), ACSA and MITRE, Williamsburg, VA*, 2002.

[49] Peter Mell, Karen Scarfone, and Sasha Romanosky. A complete guide to the common vulnerability scoring system version 2.0. In *Published by FIRST-Forum of Incident Response and Security Teams*, pages 1–23, 2007.

[50] MITRE. Common platform enumeration website. https://cpe.mitre.org/about/.

[51] MITRE. Common vulnerabilities and exposures website.

[52] MITRE. Common weakness enumeration website.

[53] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. Dismal code: Studying the evolution of security bugs. In *Proceedings of the LASER Workshop*, pages 37–48, 2013.

[54] Elizabeth Nichols, Gunnar Peterson, et al. A metrics framework to drive application security improvement. *Security & Privacy, IEEE*, 5(2):88–91, 2007.

[55] NIST. National vulnerability database website. https://nvd.nist.gov/home.cfm.

[56] OWASP. Types of application security metrics, June 2009.

[57] Top OWASP. Top 10–2013. *The Ten Most Critical Web Application Security Risks*, 2013.

[58] Donn B Parker. Toward a new framework for information security. *FLY*, page 501, 2002.

[59] Shirley C Payne. A guide to security metrics. *SANS Institute Information Security Reading Room*, 2006.

[60] Cynthia K. Riemenschneider and Bill C. Hardgrave. Explaining software developer acceptance of methodologies: A comparison of five theoretical models. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28(12):1135–1145, December 2002.

[61] Reijo Savola. A novel security metrics taxonomy for r&d organisations. In *ISSA*, volume 8, pages 379–390, 2008.

[62] Riccardo Scandariato, Bart De Win, and Wouter Joosen. Towards a measuring framework for security properties of software. In *Proceedings of the 2nd ACM workshop on Quality of protection*, pages 27–30. ACM, 2006.

[63] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys (CSUR)*, 44(3):11, 2012.

[64] Haihao Shen, Jianhong Fang, and Jianjun Zhao. Efindbugs: Effective error ranking for findbugs. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 299–308. IEEE, 2011.

[65] Verizon Enterprise Solutions. Verizon 2014 data breach investigations report. *verizon. com*, 2014.

[66] Verizon Enterprise Solutions. Verizon 2015 data breach investigations report. Technical report, Verizon Enterprise, 2015.

[67] Marianne Swanson. *Security metrics guide for information technology systems*. National Institute of Standards and Technology, Technology Administration, US Department of Commerce, 2003.

[68] Symantec. Internet security threat report. Technical report, Symantec Corporation, 2014.

[69] Symantec. Internet security threat report. Technical report, Symantec Corporation, 2015.

[70] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[71] Rahul Telang and Sunil Wattal. Impact of software vulnerability announcements on the market value of software vendors-an empirical investigation. *Available at SSRN 677427*, 2005.

[72] Jay-Evan J Tevis and John A Hamilton. Methods for the prevention, detection and removal of software security vulnerabilities. In *Proceedings of the 42nd annual Southeast regional conference*, pages 197–202. ACM, 2004.

[73] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *Security & Privacy, IEEE*, 3(6):81–84, 2005.

[74] Rayford B Vaughn Jr, Ronda Henning, and Ambareen Siraj. Information assurance measures and metrics-state of practice and proposed taxonomy. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2003.

[75] Viswanath Venkatesh and Fred D. Davis. A theoretical extension of the technology acceptance model: Four longitudinal field studies. *Management Science*, 46(2):186–204, February 2000.

[76] Antonio Vetro, Maurizio Morisio, and Marco Torchiano. An empirical validation of findbugs issues related to defects. In *Evaluation & Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*, pages 144–153. IET, 2011.

[77] Antonio Vetro, Marco Torchiano, and Maurizio Morisio. Assessing the precision of findbugs by mining java projects developed at a university. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 110–113. IEEE, 2010.

[78] Jeffrey Voas, Anup Ghosh, Gary McGraw, FACF Charron, and KAMK Miller. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In *Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on*, pages 250–263. IEEE, 1996.

[79] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. Comparing bug finding tools with reviews and tests. *Lecture Notes in Computer Science*, 3502:40–55, 2005.

[80] Lingyu Wang, Tania Islam, Tao Long, Anoop Singhal, and Sushil Jajodia. An attack graph-based probabilistic security metric. In *Data and applications security XXII*, pages 283–296. Springer, 2008.

[81] Karl E Wiegers. Seven truths about peer reviews. *Cutter IT Journal*, 15(7):31–37, 2002.

[82] Haiyun Xu, Jeroen Heijmans, and Joost Visser. A practical model for rating software security. In *SERE (Companion)*, pages 231–232, 2013.

Appendix A

Bug Patterns Categorization

The bug patterns, of FindBugs and Find Security Bugs, contained in the Latent and Acute Issues metrics are given below. More information regarding each bug pattern can be found in the static analysis tools websites [5], [4].

FindBugs Category	**Bug Pattern**
Malicious Code	DP CREATE CLASSLOADER INSIDE DO PRIVILEGED
	DP DO INSIDE DO PRIVILEGED
	EI EXPOSE REP
	EI EXPOSE REP2
	FI PUBLIC SHOULD BE PROTECTED
	EI EXPOSE STATIC REP2
	MS CANNOT BE FINAL
	MS EXPOSE REP
	MS FINAL PKGPROTECT
	MS MUTABLE ARRAY
	MS MUTABLE COLLECTION
	MS MUTABLE COLLECTION PKGPROTECT
	MS MUTABLE HASHTABLE

FindBugs Category	**Bug Pattern**
Malicious Code	MS OOI PKGPROTECT
	MS PKGPROTECT
	MS SHOULD BE FINAL
	MS SHOULD BE REFACTORED TO BE FINAL
Find Security Bugs	Untrusted Servlet Parameter
	Untrusted Content-Type Header
	Untrusted Hostname Header
	Untrusted Session Cookie Value
	Untrusted Query String
	HTTP Headers Untrusted
	Untrusted Referer Header
	Untrusted User-Agent Header
	Found JAX-WS SOAP Endpoint
	Found JAX-RS REST Endpoint
	Found Tapestry Page
	Found Wicket Page
	Regex DOS (ReDOS)
	Found Struts 1 Action
	Found Struts 2 Endpoint
	Found Spring Endpoint
	Bad hexadecimal concatenation
	External File Access (Android)
	Broadcast (Android)
	World Writable File (Android)
	WebView with Geolocation Activated (Android)
	WebView with JavaScript Enabled (Android)
	WebView with Javascript Interface (Android)

Table A.1: Latent Issues Bug Patterns.

FindBugs Category	**Bug Pattern**	**OWASP Top 10 Entry**
Security	DMI CONSTANT DB PASSWORD	A2 Broken Authentication and Session Management
	DMI EMPTY DB PASSWORD	A2 Broken Authentication and Session Management
	HRS REQUEST PARAMETER TO COOKIE	A3 Cross-Site Scripting (XSS)
	HRS REQUEST PARAMETER TO HTTP HEADER	A3 Cross-Site Scripting (XSS)
	PT ABSOLUTE PATH TRAVERSAL	A4 Insecure Direct Object References
		A7 Missing Function Level Access Control
	PT RELATIVE PATH TRAVERSAL	A4 Insecure Direct Object References
		A7 Missing Function Level Access Control
	SQL NONCONSTANT STRING PASSED TO EXECUTE	A1 Injection
	SQL PREPARED STATEMENT GENERATED FROM NONCONSTANT STRING	A1 Injection
	XSS REQUEST PARAMETER TO JSP WRITER	A3 Cross-Site Scripting (XSS)
	XSS REQUEST PARAMETER TO SEND ERROR	A3 Cross-Site Scripting (XSS)
	XSS REQUEST PARAMETER TO SERVLET WRITER	A3 Cross-Site Scripting (XSS)

FindBugs Category	**Bug Pattern**	**OWASP Top 10 Entry**
Find Security Bugs	Potentially Sensitive Data in Cookie	A6 Sensitive Data Exposure
	Potential Path Traversal (File Read)	A4 Insecure Direct Object References
		A7 Missing Function Level Access Control
	Potential Path Traversal (File Write)	A4 Insecure Direct Object References
		A7 Missing Function Level Access Control
	Potential Command Injection	A1 Injection
	FilenameUtils Not Filtering Null Bytes	A6 Sensitive Data Exposure
	TrustManager Implementation Empty	A6 Sensitive Data Exposure
	MessageDigest Is Weak	A6 Sensitive Data Exposure
	MessageDigest Is Custom	A6 Sensitive Data Exposure
	Tainted Filename Read	A4 Insecure Direct Object References
		A7 Missing Function Level Access Control
	XML Parsing Vulnerable to XXE (SAXParser)	A6 Sensitive Data Exposure
	XML Parsing Vulnerable to XXE (XMLReader)	A6 Sensitive Data Exposure
	XML Parsing Vulnerable to XXE (DocumentBuilder)	A6 Sensitive Data Exposure
	Potential XPath Injection	A1 Injection
	Potential SQL/HQL Injection (Hibernate)	A1 Injection
	Potential SQL/JDOQL Injection (JDO)	A1 Injection

FindBugs Category	Bug Pattern	OWASP Top 10 Entry
	Potential SQL/JPQL Injection (JPA)	A1 Injection
	Potential LDAP Injection	A1 Injection
	Potential code injection when using Script Engine	A1 Injection
	Potential code injection when using Spring Expression	A1 Injection
	Hazelcast Symmetric Encryption	A6 Sensitive Data Exposure
	NullCipher Unsafe	A6 Sensitive Data Exposure
	Unencrypted Socket	A6 Sensitive Data Exposure
	DES / DESede Unsafe	A6 Sensitive Data Exposure
	RSA NoPadding Unsafe	A6 Sensitive Data Exposure
	Hard Coded Password	A2 Broken Authentication and Session Management
	Struts Form Without Input Validation	A1 Injection
	XSSRequestWrapper is Weak XSS Protection	A3 Cross-Site Scripting (XSS)
	Blowfish Usage with Weak Key Size	A6 Sensitive Data Exposure
	RSA Usage with Weak Key Size	A6 Sensitive Data Exposure
	Unvalidated Redirect	A10 Unvalidated Redirects and Forwards
	Potential XSS in Servlet	A3 Cross-Site Scripting (XSS)
	XMLDecoder usage	A1 Injection

FindBugs Category	**Bug Pattern**	**OWASP Top 10 Entry**
	Static IV	A6 Sensitive Data Exposure
	ECB Mode Unsafe	A6 Sensitive Data Exposure
	Cipher is Susceptible to Padding Oracle	A6 Sensitive Data Exposure
	Cipher With No Integrity	A6 Sensitive Data Exposure
	Usage of ESAPI Encryptor	A6 Sensitive Data Exposure

Table A.2: Acute Issues Bug Patterns.

Appendix B

Latent Issues Rating Method

B.1 First level thresholds derivation

In order to identify the benchmarked risk thresholds, used in the first level of aggregation (Figure 5.5), we have followed the approach of Alves et. al. [11], as illustrated in figure B.1. The six steps of the methodology were:

1. **metrics extraction**: the number of latent issues, for each file of our dataset of systems, was identified. Additionally, the size of the files was calculated, to be used as the files' weight.

2. **weight ratio calculation**: for each file we computed its weight percentage within its system, i.e. we divided the file's size to the overall size of the system. For each system, the sum of the all its files WeightRatio must be 100%.

3. **file aggregation**: we aggregate the weightRatios of all files per metric value. For each system, therefore, the sum of the weightRatios of all metric values must be 100%.

4. **system aggregation**: we normalize the weights for the number of systems and then aggregate the weights for all systems. Normalization ensures that the sum of all the metric values would result to 100% of the overall size of the dataset.

5. **weight ratio aggregation**: we order the metric values in ascending weight and take the maximal metric value that represents 1%, 2%,....,100%.

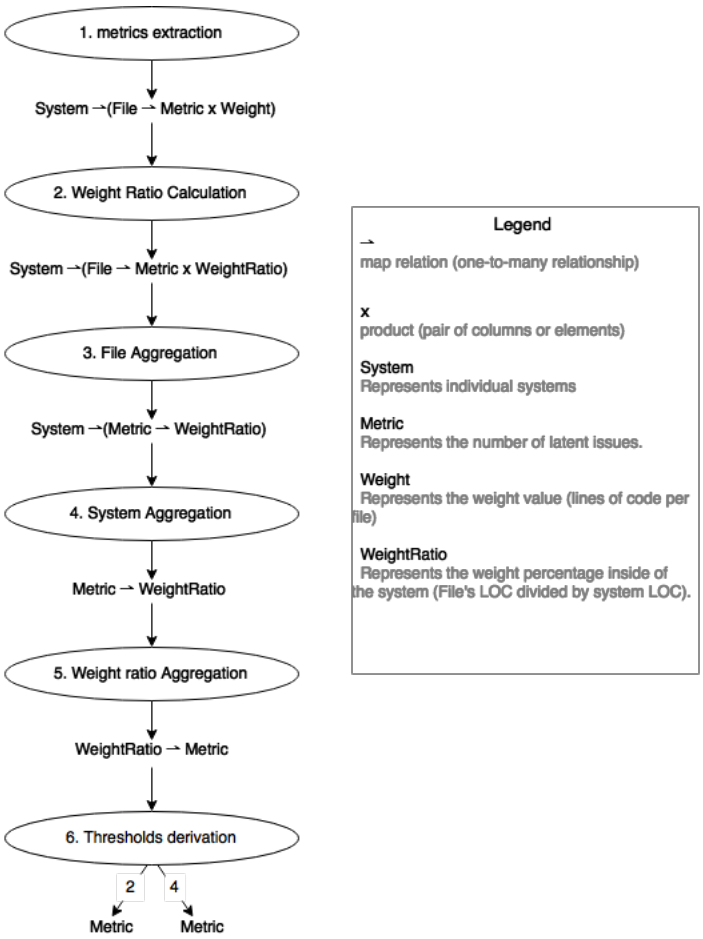


Figure B.1: Identification of Risk Thresholds methodology.

6. **thresholds derivation**: thresholds are derived by choosing the percentage of the overall code we want to represent.

The derived thresholds are 2 and 4 respectively, representing 70% and 88% of the vulnerable code. That means that we treat 70% of the overall vulnerable code (code that contains 1 or 2 latent issues per file) as code of moderate risk, the following 18% (which contains 3 or 4 latent issues per file) as code of high risk and the rest as code of very high risk. That aligns to the 70-10-10-10 principle applied at SIG with some changes due to the size of the dataset and the limited number of bugs per file.

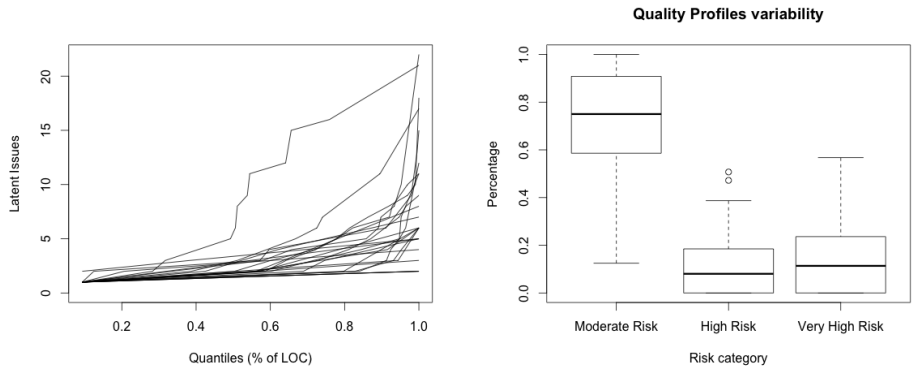
In order to choose these thresholds the variability between systems was taken into account. As it can be observed in the Quantile plot of figure B.2a, systems differentiate the most in the last quantiles (quantiles higher than

or to 70%). Choosing a quantile for which there is very low variability will result in a threshold which would not allow to distinguish quality between systems. On the other hand, choosing a quantile for which there is too much variability might fail to identify code in many systems [11].

To investigate if the thresholds are representative of those percentages of code in the dataset, we computed the quality profiles for each system. Ideally, for each system we expect to identify around 70% of the vulnerable code in the moderate risk category, 18% in the high risk category and 12% in the very high risk category. Figure B.2b depicts a box plot for all systems per risk category. The x-axis represents the three risk categories and the y-axis represents the percentage of volume of each system per risk category. The size of the box is the interquartile range (IQR) and is a measure of variability. The vertical lines indicate the lowest/maximum value within 1.5 IQR. The crosses in the charts represent systems whose risk category is higher than 1.5 IQR. In the moderate risk category, we observe large variability which is explained because it is considering a large percentage of code. In the very high risk category, additionaly, we observe that the variability is increasing compared to the variability of the high risk category. This increase was also expected since we have observed that the variability of the metric is higher for the last quantiles of the metric. Moreover, there are only two crosses in the high risk category, which means that most of the systems are represented by the box plot. Finally, looking in the median of the observations for all risk categories (the middle of the boxes), we observe that indeed the median are near to our expectation. For the moderate risk category the median is near 70%, whereas for the high risk and very high risk the median are approaching 15%. The variation, from the proposed median of 18% and 12% respectively, can be explained by the limited systems in our dataset and the few discrete values that most of the latent issues per file have, as we identified in the statistical analysis of the previous subsection. Summarizing, the box plot shows that the derived thresholds allow to observe differences between systems in our dataset.

B.2 Second level thresholds derivation

In order to acquire the set of second level thresholds, which are used for the aggregation of risk profiles to ratings, we applied the proposed calibration algorithm of Alves et. al. [10]. The algorithm works by taking the risk profiles for all the systems in our dataset and searching for the minimum thresholds that can divide those systems based on the given distribution of 5% - 30 % - 30% - 30% - 5% of systems in each star category.



(a) Distribution of latent issues in the dataset of systems.

(b) Quality profiles variability.

Figure B.2: Defining the risk thresholds.

Appendix C

Effect of filters on the precision of the Dependency-check

In this Appendix we present the results of the small scale experiment conducted on our dataset of systems. More specifically, we manually checked the reported vulnerable dependencies of 9 systems to determine whether they were true or false positives and verify whether there is an improvement after the application of the filters.

The columns of the tables provide:

1. the name of the system,
2. the maven identifier of the dependency,
3. the CPE identifier of the dependency,
4. the confidence level produced by Dependency Check (of whether the reported vulnerability is a true or false positive),
5. the confidence level after the application of the filters and,
6. whether, after manual examination, the dependency identification is a true or false positive.

Table C.1: Effect of filters on the precision of the Dependency-check.

System	Maven Identifier	CPE Identifier	Confidence	Confidence After	T/F Positive
H20	org.apache.httpcomponents:httpclient:4.1.1	cpe:/a:apache:httpclient:4.1.1	Highest	Highest	T
	org.apache.hadoop:hadoop-hdfs:0.23.4	cpe:/a:apache:hadoop:0.23.4	Highest	Highest	T
	-	cpe:/a:apache:hadoop:1.0.0	Highest	Highest	T
	org.apache.hadoop:hadoop-mapreduce-client-core:2.4.1-mapr-1408	cpe:/a:apache:hadoop:2.4.1	Highest	Highest	T
	org.jogamp.gluegen:gluegen-rt:2.0.2-rc12	cpe:/a:jogamp:jogamp:2.0.2.rc12	Low	Low	F
	org.jogamp.jocl:jocl:2.0.2-rc12	cpe:/a:jogamp:jogamp:2.0.2.rc12	Low	Low	F
	org.apache.poi:poi:3.8	cpe:/a:apache:poi:3.8	Highest	Highest	T
	-	cpe:/a:apache:zookeeper:3.3.2	Low	Low	T
	commons-httpclient:commons-httpclient:3.1	cpe:/a:apache:commons-httpclient:3.1	Low	High	T
Java-Game-Server	com.adobe.blazeds:blazeds-core:3.2.0.3978	cpe:/a:adobe:blazeds:3.2.0.3978	Low	Low	T
Jenkins	org.apache.jackrabbit:jackrabbit-jcr-commons:1.5.0	cpe:/a:apache:jackrabbit:1.5.0	Highest	Highest	T
	org.apache.httpcomponents:httpclient:4.2.3	cpe:/a:apache:httpclient:4.2.3	Highest	Highest	T
	org.eclipse.aether:aether-spi:0.9.0	cpe:/a:eclipse:eclipse-ide:0.9.0.m2	Low	Low	F
	org.apache.httpcomponents:httpclient:4.2.3	cpe:/a:apache:httpclient:4.2.3	Highest	Highest	T
	org.acegisecurity:acegi-security:1.0.7	cpe:/a:acegisecurity:acegi-security:1.0.7	Highest	Highest	T
	commons-httpclient:commons-httpclient:3.1	cpe:/a:apache:commons-httpclient:3.1	Low	High	T

Table C.2: Effect of filters on the precision of the Dependency-check.

System	Maven Identifier	CPE Identifier	Confidence	Confidence After	T/F Positive
Jenkins	org.jruby.ext.posix:jna-posix:1.0.3-jenkins-1	cpe:/a:jruby:jruby:1.0.3	Highest	Highest	F
	javax.mail:mail:1.4.4	cpe:/a:sun:javamail:1.4.4	Low	Low	T
	org.springframework:spring-core:2.5.6.SEC03	cpe:/a:vmware:springsource-spring-framework:2.5.6.sec03	Low	High	T
	org.springframework:spring-dao:1.2.9	cpe:/a:vmware:springsource-spring-framework:1.2.9	Low	High	T
	org.springframework:spring-core:2.5.6.SEC03	cpe:/a:vmware:springsource-spring-framework:2.5.6.sec03	Low	High	T
	org.springframework:spring-jdbc:1.2.9	cpe:/a:vmware:springsource-spring-framework:1.2.9	Low	High	T
Keybox	com.google.code.gson:gson:2.3.1	cpe:/a:google:v8:2.3.1	Low	Low	F
	org.apache.struts:struts2-core:2.3.20	cpe:/a:apache:struts:2.3.20	Low	Low	T
	org.apache.struts.xwork:xwork-core:2.3.20	cpe:/a:apache:struts:2.3.20	Low	Low	F
	org.apache.struts:struts2-core:2.3.20	cpe:/a:apache:struts:2.3.20	Low	Low	T
LibGDX	org.apache.httpcomponents:httpclient:4.3.1	cpe:/a:apache:httpclient:4.3.1	Highest	Highest	T
	org.apache.httpcomponents:httpmime:4.3.1	cpe:/a:apache:httpclient:4.3.1	Highest	Highest	T
	-	cpe:/a:freetype:freetype:-	Low	Low	F
	-	cpe:/a:freetype:freetype:-	Low	Low	F
	com.badlogicgames.gdx:gdx-freetype-platform:1.5.4-SNAPSHOT	cpe:/a:freetype:freetype:1.5.4	Low	Low	F
	com.badlogicgames.gdx:gdx-freetype:1.5.4-SNAPSHOT	cpe:/a:freetype:freetype:1.5.4	Low	Low	F
	-	cpe:/a:freetype:freetype:-	Low	Low	F
	-	cpe:/a:freetype:freetype:-	Low	Low	F

Table C.3: Effect of filters on the precision of the Dependency-check.

System	Maven Identifier	CPE Identifier	Confidence	Confidence After	T/F Positive
Mahout	commons-httpclient:commons-httpclient:3.1	cpe:/a:apache:commons-httpclient:3.1	Low	Highest	T
	org.glassfish.grizzly:grizzly-framework:2.1.2	cpe:/a:oracle:glassfish:2.1.2	Low	Low	F
	org.glassfish.grizzly:grizzly-http-server:2.1.2	cpe:/a:oracle:glassfish:2.1.2	Low	Low	T
	org.glassfish.grizzly:grizzly-rcm:2.1.2	cpe:/a:oracle:glassfish:2.1.2	Low	Low	T
	org.apache.hadoop:hadoop-mapreduce-client-core:2.2.0	cpe:/a:apache:hadoop:2.2.0	Highest	Highest	T
	org.mortbay.jetty:jetty-util:6.1.26	cpe:/a:jetty:jetty:6.1.26	Low	Low	T
	org.apache.solr:solr-commons-csv:3.5.0	cpe:/a:apache:solr:1.0.0	Highest	Highest	T
	org.apache.zookeeper:zookeeper:3.4.5	cpe:/a:apache:zookeeper:3.4.5	Low	Low	T
	io.netty:netty:3.6.6.Final	cpe:/a:netty-project:netty:3.6.6	Highest	Highest	T
	org.apache.hadoop:hadoop-annotations:2.2.0	cpe:/a:apache:hadoop:2.2.0	Highest	Highest	T
	org.apache.hadoop:hadoop-auth:2.2.0	cpe:/a:apache:hadoop:2.2.0	Highest	Highest	T
	org.apache.hadoop:hadoop-client:2.2.0	cpe:/a:apache:hadoop:2.2.0	Highest	Highest	T
	org.apache.hadoop:hadoop-common:2.2.0	cpe:/a:apache:hadoop:2.2.0	Highest	Highest	T
	org.apache.mahout:mahout-h2o:1.0-SNAPSHOT	cpe:/a:apache:apache-test:1.0	Low	Low	F
	org.apache.mahout:mahout-math:1.0-SNAPSHOT	cpe:/a:apache:apache-test:1.0	Low	Low	F
	org.apache.mahout:mahout-mrlegacy:1.0-SNAPSHOT	cpe:/a:apache:apache-test:1.0	Low	Low	F
	net.sf.py4j:py4j:0.8.2.1	cpe:/a:python:python:0.8.2.1	Low	Low	F
	org.apache.curator:curator-framework:2.4.0	cpe:/a:apache:zookeeper:2.4.0	Low	Low	F
	org.apache.geronimo.specs:geronimo-jta-1.1-spec:1.1.1	cpe:/a:apache:geronimo:1.1.1	Highest	Highest	T

Table C.4: Effect of filters on the precision of the Dependency-check.

System	Maven Identifier	CPE Identifier	Confidence	Confidence After	T/F Positive
Internal System	com.atlassian.security:atlassian-cookie-tools:3.0	cpe:/a:vmware:springsource-spring-security:3.0.0	Highest	Low	F
	commons-httpclient:commons-httpclient:3.1	cpe:/a:apache:commons-httpclient:3.1	Low	Highest	T
	org.javassist:javassist:3.18.1-GA	cpe:/a:springsource:spring-framework:3.18.1	Low	Low	F
	org.glassfish.jersey.core:jersey-common:2.13	cpe:/a:springsource:spring-framework:2.13	Low	Low	F
	mysql:mysql-connector-java:5.1.32	cpe:/a:mysql:mysql:5.1.32-bzr	Highest	Highest	T
	taglibs:standard:1.1.2	cpe:/a:apache:standard-taglibs:1.1.2	Low	Low	T
	org.apache.struts:struts2-core:2.3.16.3	cpe:/a:apache:struts:2.3.16.3	Highest	Highest	T
Ninja	javax.mail:mail:1.4.5	cpe:/a:sun:javamail:1.4.5	Low	Low	T
OkHttp	io.airlift:airline:0.6	cpe:/a:git:git:0.6	Highest	Low	F
	com.squareup.okhttp:okcurl:2.3.0-SNAPSHOT	cpe:/a:curl:curl:2.3.0	Low	Low	F
	com.squareup.okhttp:okhttp-apache:2.3.0-SNAPSHOT	cpe:/a:apache:httpclient:2.3.0	Low	High	T

Appendix D

Longitudinal Analysis of systems

Appendix D presents the systems used in the longitudinal analysis of Chapter 7. Table D.1 provides the systems, their versions and the release dates of the versions. Table D.2 provides the "Acute" and "Latent Issues" ratings for each version of the systems. Furthermore, Table D.3 shows the size difference (in terms of lines of code) between the subsequent versions of the systems.

System	Version	Release Date
Guacamole	0.8.2	15/07/2013
	0.9.0	28/03/2014
	0.9.1	23/05/2014
	0.9.2	21/07/2014
	0.9.3	30/09/2014
	0.9.4	07/01/2015
	0.9.6	31/03/2015
	0.9.7	11/06/2015
Webbit	0.2.13	01/11/2011
	0.4.6	23/02/2012
	0.4.11	27/06/2012
	0.4.14	13/07/2012
	0.4.15	26/04/2013
	0.4.16	17/07/2014
	0.4.18	14/11/2014
	0.4.19	08/01/2015
Ninja	1.6.0	07/08/2013
	2.2.0	13/11/2013
	2.5.0	06/01/2014
	3.1.4	06/04/2014
	3.3.0	31/07/2014
	4.0.0	03/11/2014
	4.0.68	27/02/2015
	5.1.2	08/06/2015
OkHttp	1.2.0	12/08/2013
	1.3.0	12/01/2014
	1.5.0	07/03/2014
	2.0.0	21/06/2014
	2.1.0	12/11/2014
	2.2.0	31/12/2014
	2.3.0	17/03/2015
	2.4.0	23/05/2015
Fluent-Http	1.28	01/02/2014
	2.0	20/05/2014
	2.0.9	19/07/2014
	2.14	17/10/2014
	2.50	13/01/2015
	2.80	10/03/2015
	2.90	01/04/2015
	2.98	08/06/2015

Table D.1: Systems examined in the longitudinal analysis.

System	Version	Latent Rating	Acute Rating
Guacamole	0.8.2	4.4138	3.0
	0.9.0	4.4458	3.0
	0.9.1	4.3321	3.0
	0.9.2	4.3326	3.0
	0.9.3	4.3407	3.0
	0.9.4	4.0362	3.0
	0.9.6	3.5995	3.0
	0.9.7	3.4536	3.25
Webbit	0.2.13	4.0840	4.0
	0.4.6	3.2712	4.0
	0.4.11	3.2345	4.0
	0.4.14	3.2063	4.0
	0.4.15	2.5288	3.5
	0.4.16	2.4983	3.5
	0.4.18	2.5010	3.5
	0.4.19	2.4983	3.5
Ninja	1.6.0	2.8715	5.5
	2.2.0	2.9632	5.5
	2.5.0	2.9048	5.5
	3.1.4	3.0066	5.5
	3.3.0	3.5957	5.5
	4.0.0	3.6833	5.5
	4.0.68	3.5406	5.5
	5.1.2	3.4289	5.5
OkHttp	1.2.0	3.3081	3.5
	1.3.0	3.3314	3.5
	1.5.0	4.0291	3.0
	2.0.0	4.1092	3.0
	2.1.0	4.1226	3.0
	2.2.0	4.1339	3.0
	2.3.0	4.0296	3.0
	2.4.0	4.2227	3.0
Fluent-Http	1.28	2.6393	4.0
	2.0	2.3742	4.0
	2.0.9	2.4500	4.0
	2.14	2.3997	4.0
	2.50	2.4612	4.0
	2.80	3.0699	4.0
	2.90	3.0823	4.0
	2.98	3.0529	4.0

Table D.2: Ratings of the systems used in the analysis.

System	**Version**	**Size Difference with previous version** *(Lines of Code)*
Guacamole	0.8.2	0
	0.9.0	2797
	0.9.1	63
	0.9.2	13
	0.9.3	236
	0.9.4	1020
	0.9.6	1551
	0.9.7	1227
Webbit	0.2.13	0
	0.4.6	975
	0.4.11	257
	0.4.14	38
	0.4.15	187
	0.4.16	-264
	0.4.18	-1
	0.4.19	1
Ninja	1.6.0	0
	2.2.0	747
	2.5.0	426
	3.1.4	1048
	3.3.0	754
	4.0.0	1827
	4.0.68	9396
	5.1.2	-8166
OkHttp	1.2.0	0
	1.3.0	1083
	1.5.0	2725
	2.0.0	305
	2.1.0	579
	2.2.0	1338
	2.3.0	341
	2.4.0	1857
Fluent-Http	1.28	0
	2.0	1754
	2.0.9	1443
	2.14	-818
	2.50	734
	2.80	505
	2.90	131
	2.98	-89

Table D.3: Size difference between subsequent versions.