Measuring software security from the design of software

UNIVERSITY OF TURKU
Department of IT
Master’s thesis in technology
Marko Saarela
February 2016

UNIVERSITY OF TURKU
Department of Information Technology

MARKO SAARELA: Measuring software security from the design of software

Master's thesis in technology, 73 p, 5 p of appendixes
Software Engineering
February 2016

The vast majority of our contemporary society owns a mobile phone, which has resulted in a dramatic rise in the amount of networked computers in recent years. Security issues in the computers have followed the same trend and nearly everyone is now affected by such issues. How could the situation be improved? For software engineers, an obvious answer is to build computer software with security in mind.

A problem with building software with security is how to define secure software or how to measure security. This thesis divides the problem into three research questions. First, how can we measure the security of software? Second, what types of tools are available for measuring security? And finally, what do these tools reveal about the security of software? Measuring tools of these kind are commonly called metrics.

This thesis is focused on the perspective of software engineers in the software design phase. Focus on the design phase means that code level semantics or programming language specifics are not discussed in this work. Organizational policy, management issues or software development process are also out of the scope. The first two research problems were studied using a literature review while the third was studied using a case study research. The target of the case study was a Java based email server called Apache James, which had details from its changelog and security issues available and the source code was accessible.

The research revealed that there is a consensus in the terminology on software security. Security verification activities are commonly divided into evaluation and assurance. The focus of this work was in assurance, which means to verify one's own work. There are 34 metrics available for security measurements, of which five are evaluation metrics and 29 are assurance metrics.

We found, however, that the general quality of these metrics was not good. Only three metrics in the design category passed the inspection criteria and could be used in the case study. The metrics claim to give quantitative information on the security of the software, but in practice they were limited to evaluating different versions of the same software. Apart from being relative, the metrics were unable to detect security issues or point out problems in the design. Furthermore, interpreting the metrics' results was difficult.

In conclusion, the general state of the software security metrics leaves a lot to be desired. The metrics studied had both theoretical and practical issues, and are not suitable for daily engineering workflows. The metrics studied provided a basis for further research, since they pointed out areas where the security metrics were necessary to improve whether verification of security from the design was desired.

Keywords: computer security, software security, software design, evaluation, assurance, measuring security, metrics

TURUN YLIOPISTO
Informaatioteknologian laitos

MARKO SAARELA: Ohjelmistoturvallisuuden mittaaminen ohjelman suunnitelmasta

Diplomityö, 73 s., 5 liites.
Ohjelmistotekniikka
Helmikuu 2016

Lähes jokaisella on matkapuhelin näinä päivinä, mikä samalla tarkoittaa tietoverkkoon liittyneiden tietokoneiden määrän nousseen dramaattisesti viime vuosina. Tietokoneiden turvallisuusongelmat ovat myös seuranneet tätä nousevaa trendiä ja ongelmat koskettavat lähes jokaista. Mikä olisi avuksi tässä tilanteessa? Ohjelmistoinsinööreille vastaus on selvä: ohjelmat tulee kehittää alusta alkaen turvallisuus huomioiden.

Tässä työssä tutkimusongelma on jaettu kolmeen tutkimuskysymykseen. Ensimmäisenä selvitetään miten ohjelmiston turvallisuutta voidaan mitata. Toisena selvitetään mitä työkaluja turvallisuuden mittaukseen on olemassa. Kolmantena tutkitaan mitä nämä työkalut oikein kertovat ohjelmiston turvallisuudesta. Mittaustyökaluista tässä yhteydessä käytetään yleisesti termiä metriikat.

Tämä tutkimus tarkastelee ongelmaa ohjelmistoinsinöörien näkökulmasta ohjelmiston suunnitteluvaiheessa. Suunnitteluvaiheeseen keskittyminen tarkoittaa, että kooditason semantiikan tai ohjelmointikielten yksityiskohtien tarkastelu jätetään työn ulkopuolelle. Myöskään organisaation toimintatapojen, johtamisen tai ohjelmistokehitysprosessien tarkastelu eivät kuulu tutkimukseen. Kahteen ensimmäiseen ongelmaan käytettiin menetelmänä kirjallisuuskatsausta. Katsauksen jälkeen viimeistä ongelmaa tutkittiin tapaustutkimuksen avulla. Tapaustutkimuksen kohteena oli Java-pohjainen sähköpostipalvelin nimeltään Apache James, josta oli saatavilla muutosloki, tietoja haavoittuvuuksista sekä pääsy lähdekoodiin.

Tutkimuksen tuloksena selvisi, että ohjelmistoturvallisuuden englanninkielisestä terminologiasta on olemassa jonkinlainen yksimielisyys. Turvallisuuden arviointitoiminnot jaetaan yleisesti arviointiin ja varmistamiseen. Tämä työ keskittyi varmistamiseen, joka siis tarkoittaa ohjelmiston rakentajan oman työn turvallisuuden varmistamista. Työssä löydettiin yhteensä 34 metriikkaa, joista 5 keskittyi arviointiin ja 29 varmistamiseen.

Metrikoiden yleinen laatu oli heikko. Ainoastaan kolme suunnittelukategorian metriikkaa läpäisi tarkastelukriteerit ja päätyi käytettäväksi tapaustutkimukseen. Metriikat väittävät antavansa kvantitatiivista tietoa ohjelmiston turvallisuudesta. Kuitenkin ne ovat suhteellisia ja tämän lisäksi ne eivät kyenneet löytämään turvallisuushaavoittuvuuksia tai osoittamaan muitakaan ongelmia ohjelman mallissa.

Yhteenvetona metriikoiden yleinen tila jättää paljon toivomisen varaa. Tutkimuksessa tarkastelluissa metriikoissa oli sekä teoreettisia ongelmia että soveltamisongelmia. Metriikoiden arviointi tarjoaa kuitenkin pohjan jatkotutkimukselle, sillä jos turvallisuutta halutaan metriikoilla arvioida, niin tarkastelussa havaitut kehityskohteet on huomioitava.

Asiasanat: tietoturvallisuus, ohjelmistoturvallisuus, ohjelmistosuunnittelu, arviointi, varmistaminen, turvallisuuden mittaaminen, metriikat

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Motivation

At present, software security is an important topic in light of the dramatic rise in the number of networked computers, such as mobile phones, even compared to the situation one decade ago. At the same time, detected security issues have increased dramatically and because almost everyone has a mobile phone, they are connected to almost everyone on a daily basis. How can security issues be avoided when buying new antivirus software is not a solution?

For software engineers the obvious answer should be to build the software with security in mind in the first place. One of the problems in building secure software is how to measure the security of the software. In other words, how to define secure software and how to measure that the software in question meets this definition?

1.2 Scope

This thesis deals with the measurement of software security from the perspective of a software architect or developer. The focus is on the software development lifecycle design phase and software design. The software development process is not the focus of this thesis, but it is discussed lightly along with the topic of computer security in order to define what is meant by the term ‘software security’.

This thesis is not focusing on organizational policy, project management issues, code level semantical issues, or programming language specifics.

1.3 Objectives and research questions

The main objective of this thesis is to see what security metrics are there available for measuring the security of software design and whether those metrics actually reveal anything meaningful. One objective is also to inspect the usability of the metrics. Can the metrics use the common software design artifacts, such as UML charts? Are the metrics practical in terms of effort required to use them? Finally, an objective is to determine the ultimate meaning of ‘secure software’ and how is it related to software design.

The research questions in this thesis are:

1. How can we measure the security of a software?

2. What metrics are available for measuring software security?

3. What do software security metrics actually reveal about the security of the software?

From now on, the research questions will be marked as RQ1, RQ2 and RQ3.

1.4 Methods

A literature review is used to answer the first and second questions. For the first research question the purpose is to find a definition for software security and what it means to measure such things. The purpose of the second research question is to find out what kind of metrics are available, describe them, and based on the findings for the first research question, try to find out which are suitable for this study. For the third research question a case study research design is used in which selected metrics are applied to a target software. The purpose is to find out whether the metrics are able to measure anything, are feasible to use, and are they capable of revealing anything about the security.

1.5 Contents

Chapter 2 explores the problem of defining the phrase, ‘software security’. Chapter 3 briefly explains measurement theory and presents proposed metrics for software security measurement. Chapters 2 and 3 provide answers to RQ1 and RQ2. Chapters 4 and 5 provide an in-depth explanation of the two chosen metrics for the case study. Chapter 6 introduces the targeted case study and presents how the case study was conducted. Chapter 7 shows the results of the case study and interprets them. Chapter 8 analyzes and discusses what the results tell about the security metrics and measuring security.

2 WHAT IS SOFTWARE SECURITY?

2.1 Defining security

Because this thesis is focused on computer security the first step to take before measuring any security is to define what exactly is meant by ‘security’. So what is computer security? Few actually attempt to define it, even though most agree that having it is good [1]. In the scope of this thesis, computer security is defined as in [2], which starts with dividing computer security into software and application security. Other aspects of security engineering, such as physical security, are not considered while discussing computer security. The reason for this narrow scope is an important idea expressed in [2] and [3]: The central culprit of issues concerning computer security is actually software security.

There is a difference between the two classes of computer security mentioned above. Application security entails protection of software after it has already been built. The viewpoint of application security originates from a network centric approach to computer security. It includes functions such as penetration testing and is reactive in nature. Using a firewall is an example of a use of an application security technology. [2]

Many sources consider focusing only on the application security as an insufficient approach and state that security should be built into the applications from the beginning of their development process. [2] [3]

The concept that this thesis focuses on -software security- means designing, building and testing software for its security. Software security should not be confused with security software. The point of software security is to ensure that people developing software do a better job in considering security as an integral part of the software. Software security takes into account both security mechanisms (such as cryptography) and design for security (such as design that makes circumventing the use of cryptography difficult). [2]

Narrowing the scope from computer security to software security does not fulfil the original problem, but rather, transforms the question into another form. So what is software security? Security can be thought as an emergent property [2] or as a requirement for the software [1]. For a long time, software engineers have classified system requirements into functional and non-functional requirements. Security is a non-functional requirement, among others such as performance or reusability. In other words, security is a quality attribute of the system. [4]

2.2 Viewpoints to software security

Merely defining security is not enough when trying to measure software security. There are multiple viewpoints from which to look at software security, such as that of a manager, an organization or a developer. [1] [5] The viewpoint of management or an organization is usually concerned with concepts such as adherence to standards [5] or the costs of risks in comparison to the alternative of avoiding them [6]. A developer's viewpoint is usually very different from these two due to their approach and methods of working.

Software developers, such as architects, coders and testers, are usually working in a process called the software development lifecycle. This process represents all activities and work products that are necessary to develop a software system. [7] There are multiple lifecycles to choose from, and there are also specialized lifecycles which define ways to integrate software security work into the software development process. [8] The following figures describe two possible approaches to software security work. Figure 2-1 shows best practices suitable for any software development lifecycle as presented in [2]. Figure 2-2 has a shortened version of a specialized software development lifecycle used in Microsoft for security work [9].

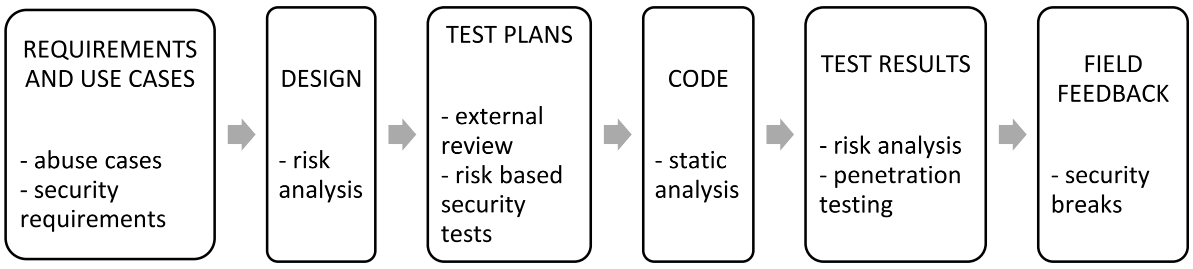


Figure 2-1: Software security best practices (adapted from [2])

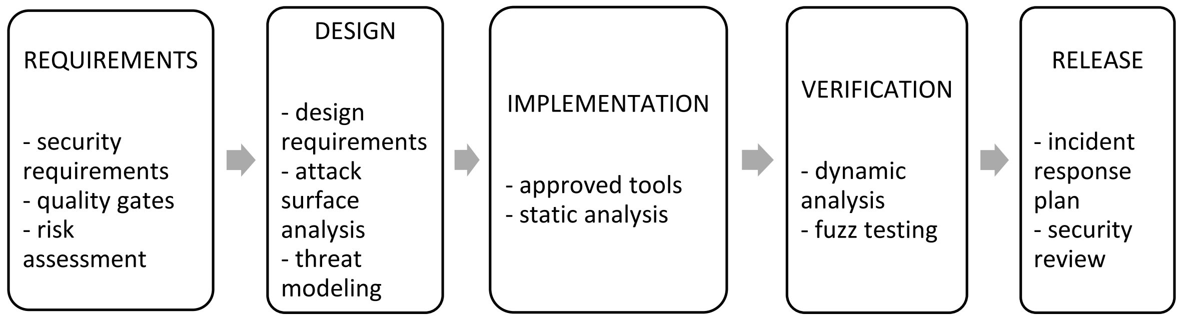


Figure 2-2: The Microsoft Security Development Lifecycle in brief (adapted from [9])

The viewpoint used in this thesis is of a software developer at the design phase, but the objective is not to study the secure software development lifecycles.

Why focus on the design phase? A valid question. Numerous studies have presented a 'cost of defect' table, such as the classical IBM study in software quality referenced in [6] or more contemporary studies [10] where the relative cost of a software defect raises exponentially during advance stages of software development lifecycle. The purpose is to demonstrate that focusing on the quality of the software as early as possible in the software development process pays off. However, this 'cost of defect' metric has been under criticism recently [11] [12] [13].

The criticism has three main points: 1. the 'cost of defect' metric actually makes quality software unfeasible economically [13]; 2. the metric does not account for differences between the size or complexity of software projects or the programming language used [13]; 3. the metric does not account for the differences between defects (e.g. whether the defect is a major showstopper or a cosmetic one [13]). Regardless of these criticisms, a focus in software quality by fixing the defects is not contested and such an approach actually does make economic sense [12]. Therefore, in order to have a useful scope for a thesis, and to keep in mind the building-in approach to software security, only the design phase is considered.

What is a security issue or defect? Security issue means that a (security) quality attribute is not met in the software. Someone who has not considered the definition of security might argue, that security has a binary value: either a software is secure or it is not [1]. This quick conclusion defeats the purpose of measuring security at all [1], so the definition of security as a quality attribute of a software is used in this thesis.

2.3 Verifying the security of a software

There is a simple and apparent problem when designing software for security: how can one verify the security that has been designed? Measuring the performance can be accomplished through simulations, but what about the security itself [14]? In other words, how can one ensure that the security requirements meet the security needs, security policy meets the requirements and the security mechanisms implement the policy [1]?

Commonly these verification activities are grouped and divided into two concepts, called assurance and evaluation, although the naming of these concepts varies across sources. The difference in these concepts is quite simple: assurance means making sure that the software works as intended, whereas evaluation means convincing other people that it works as intended [14]. This thesis uses the terms and definitions as presented in [14], but a brief look into other sources is used to clarify the situation.

International Organization for Standardization (ISO) standard 15288 on systems and software engineering processes uses the terms verification and validation [15], which differs from the terms used in this thesis. According to ISO 15288, "the purpose of the verification process is to confirm that the specified design requirements are fulfilled by the system" [15] and "the purpose of the validation process is to provide objective evidence that the services provided by a system when in use comply with stakeholders' requirements" [15]. Even though the terminology in ISO 15288 is different from this thesis, the definition and concepts used in the standard are the same.

Another source for the terms is Common Criteria [16], which uses the terms: verification, assurance and evaluation. Common Criteria defines 'verification' as: "rigorously review in detail with an independent determination of sufficiency" [16], 'assurance' as: "grounds for confidence that a target of evaluation meets the security functional requirements" [16] and 'evaluation' as: "assessment of a protection profile, a security target or a target of evaluation, against defined criteria" [16]. Common Criteria's terminology is similar to that used in this thesis, but the definitions used for assurance and evaluation are too specific to the context of Common Criteria.

Looking from the thesis' viewpoint of a software developer, the security assurance is where this thesis focuses on one out of these two. However, a brief look at the concept of security evaluation is undertaken in order to understand the matter.

The security evaluation is defined in [14] as: "the process of assembling evidence that a system meets, or fails to meet, a prescribed assurance target". The definition is a bit vague, since it overlaps with testing, but it should not be confused with testing. The purpose of evaluation is to convince a superior, a client or a court that the system is suited for the purpose that was built (i.e. that it works). The need for evaluation arises when those bearing the cost of implementing the protection are different from those who carry the risk or cost of failure, of the protection. Evaluation is usually done with a third party evaluation scheme such as the Common Criteria. [14]

Assurance is defined more precisely in [14] as an "estimate of the likelihood that a system will fail in a particular way". The estimate can be based on a number of different factors, such as the process used to develop the system, the identity of the developers, a particular technical assessment, or an introduction of deliberate flaws or experience. [14]

Assurance can focus on many things including the examination of security policy, mechanisms and/or the implementation [14]. All of these are important, but from the viewpoint of software design, this thesis focuses on the *implementation* aspect of 'assurance'. Focusing on implementation means the central topic will be whether the product has been implemented correctly with the agreed functionality and mechanisms (i.e. whether there are any technical security failures in the product). [14]

Security testing is an example of a technical assessment in [14] that is performed for security assurance. Security testing consists of reading product documentation, reviewing code and running test programs. More precisely, the process is defined as beginning with an initial assessment for architectural flaws, then to look for implementation flaws and finally to use a list of less common flaws [14]. An assessment look for architectural flaws is the process of examining the software design and looking for coding errors such as stack overflows or integer overflows. It is also possible to benefit from the experience of others and use a list of less common security flaws for the examination [14].

3 SOFTWARE SECURITY METRICS

Now that boundaries for this thesis have been established,-particularly- what is this thesis looking for and where is it related to- there is a question of how to answer such questions. What could be used to test, process or measure the software design to come up with results regarding the security of the design? Before venturing any further, the term ‘metric’ needs to be explained.

3.1 Metrics and measurement

A metric has multiple possible definitions depending on the chosen source [17] and some sources even try to avoid using the term altogether [18]. A useful metric is one that “quantitatively characterizes a property” [17], implying that there has to be a property (sometimes called the measurand [17]) to characterize.

The measurement theory also defines the terms ‘measure’, ‘measurement’ and ‘value’ [17] [18]. A measure is something that a metric needs, such as an instrument or a formula that allow the metric to be applied to related objects under inspection [17]. A measurement is a process to get the results with the measure. Finally, all measurements need to end up with a value [17].

According to [17], whenever there is a need for a measurement, all measurements end up using these five critical elements:

1) *The property* to be measured needs to be identified.
2) *A metric* needs to be defined to quantitatively characterize the property.
3) *A measure* needs to be developed that applies the metric to a target.
4) *A measurement* process needs to be designed.
5) Each measurement needs to have *a value* and an estimate of its accuracy.

However, there is an immediate problem when using concepts from the physical world and measurement theory: software is not a tangible product and therefore it does not have physical properties to measure. Two concessions are required to measure software security. First, if there are not any directly measurable properties, then a concept called ‘latent variable’ can be used. This means that a property can be estimated using some observable attributes. Second, the definition of metric can be loosened. [17]

In this thesis, the definition of *metric* is: “a consistent standard for measurement” as defined in [6]. According to [6], a good metric should be:

- *Consistently measured* without subjective criteria.
- *Cheap to gather*, preferably in an automated way.
- *Expressed as a cardinal number or percentage* instead of qualitative labels.
- *Expressed using at least one unit of measure*, such as “defects”, “hours” or “dollars”.
- Ideally, it is *contextually specific*.

This thesis uses a simple measurement process as presented in [19]:

1) Metrics need to be available.
2) A suitable metrics framework needs to be chosen and implemented.
3) Measurements need to be interpreted.

Before choosing suitable metrics frameworks, this chapter explores and presents the currently available software security metrics.

3.2 Categories of metrics

Software security metrics can be categorized in multiple ways that represent viewpoints or abstractions within the metrics. The reason for different viewpoints is obvious: a manager has very different needs for the metrics from a software developer. The categories indicate the environment where the metric works well or is designed to work while showing where the metric is most likely to fail.

The four categories presented in this section are collected from the sources of the metrics. Because most sources provide minimal examples of metrics in the categorization, the thesis author performs most categorization found here. The best attempt was made to find all related metrics for this work. The first two categories divided the metrics into two groups, but the metrics are presented from one side of the division for clarity. The division in the sections is quite obvious and presenting all of the metrics at the first categorization would cause the focus to be lost. Many presented metrics also fit into multiple categorization schemes, but when they do, they are only presented in detail for the first time they are introduced.

3.2.1 Security engineering perspective

One possible way to categorize metrics is to use the definition of verification activities, as specified in [14], which divides metrics into assurance and evaluation. Evaluation metrics verify the product or process against standards, while assurance metrics verify whether the product is built securely. The focus on this categorization is on the verification process. The following Table 3-1 presents the metrics divided into assurance and evaluation categories. However, only evaluation metrics are explained with additional detail because the assurance metrics are presented in the other categories. The metrics for evaluation criteria are chosen by the thesis author based on ideas and guidance provided by [14], [6] and [20]. This section continues by examining some of the metrics before presenting the whole Table 3-1.

Common Criteria [16], formally called Common Criteria for Information Technology Security Evaluation, is the successor to Orange book and ITSEC metrics for governmental evaluation. The Common Criteria is a three-actor evaluation scheme that includes a producer, evaluator and consumer. Its primary purpose is to verify three aspects of the product: correct definition of the requirements, correct implementation of the requirements and correct documentation [21].

Orange book [22], the Trusted Computer System Evaluation Criteria of the US Department of Defense, and its European counterpart **ITSEC** [23], Information Technology Security Evaluation Criteria, were developed due to the need to use commercial information technology systems in military use. They were abandoned as a result of troublesome evaluation practicalities, insistence on formal models for mandatory security and excessive demands for documentation [24].

NIST 800-55 [25], the Performance Measurement Guide for Information Security, is a US government guide for the development and measurement of organizational or information system level security activities. The main goal of the guide appears to be helping US government agencies to comply with legislative demands. It does not measure security as such, but it defines where security should be measured and in what ways it could be measured.

Table 3-1: Categorization of metrics into assurance and evaluation

ASSURANCE METRICS	EVALUATION METRICS
ISO 27004	Common Criteria
MBSA	Orange book
SSE-CMM	ITSEC
Security assessment framework	NIST 800-55
Security estimation framework	Component analysis
Catalog of metrics for SDLC	
CWE	
CVSS	
CMSS	
Security metrics for software systems	
Formal analysis for secure architectures	
Analyzing architectures with Bauhaus	
Measuring security article	
SEEA to ISO 26262	
Security metrics for object oriented programs	
Hierarchical assessment model	
Object oriented class design	
Object oriented multiclass design	
Attack surface analysis	
Formal constrain analysis	
OCL signature analysis	
Risk analysis for security pattern systems	
Security pattern analysis	
USIE model	
Vulnerability index in dynamic architectures	
Attack prone component prediction	
ASSM	
CCD	
Three code metrics	

Component analysis [26] is a method to evaluate the security of a software architecture based on the individual components' evaluations. The method is applicable to architectures that already have defined security requirements. The method evaluates the importance of individual components and whether their security measures fulfill the requirements.

3.2.2 Business perspective

Another way of categorization is to divide the metrics into organizational levels (enterprise level, management level) and technical level (software system property level) metrics as done in [5]. This categorization stems from differing needs regarding information between the management's organizational level and the developers' system property level. In other words, the organizational level places more focus on the economic risk management [6] or processes, while the system property focuses more on the security assurance.

Table 3-2 presents the metrics divided into organizational level and system property level categories. The organizational level metrics are explained further in this section, while system property metrics will be presented in subsequent sections.

ORGANIZATIONAL LEVEL METRICS
Common Criteria Orange book ITSEC NIST 800-55 ISO 27004 MBSA SSE-CMM Security assessment framework Security estimation framework Catalog of metrics for SDLC
SYSTEM PROPERTY LEVEL METRICS
CWE CVSS CMSS Security metrics for software systems Formal analysis for secure architectures Analyzing architectures with Bauhaus Measuring security article SEEA to ISO 26262 Security metrics for object oriented programs Hierarchical assessment model Component analysis Object oriented class design Object oriented multiclass design Attack surface analysis Formal constrain analysis OCL signature analysis Risk analysis for security pattern systems Security pattern analysis USIE model Vulnerability index in dynamic architectures Attack prone component prediction ASSM CCD Three code metrics

Table 3-2: Categorization of metrics into organizational and system property levels

ISO 27004 [27] is part of the ISO 27000 family of information security management standards. The 27004 standard provides guidance on creating and using metrics for measuring the information security management activities and systems [28].

MBSA [29], a Security Metric Based on Security Arguments, is a framework that provides a process for mapping security goals of the system stakeholders and the individual information security metrics. The fulfillment of the security goals is measured in a qualitatively using a 'degree of belief'.

SSE-CMM [30] [31], shorthand for the ISO 21827 Systems Security Engineering Capability Maturity Model, is a standard aimed at improving organizational or engineering processes related to information security. The standard defines the required processes and how to monitor and improve them.

Security assessment framework [32] is a process that attempts to help select metrics that could verify software security at the design time. No instructions on how to use the process or any security metrics are provided.

Security estimation framework [33] is a framework that defines the stages of a process, which can be used estimate security at the early stages of the software development lifecycle. Process stages are vaguely described and no security metrics are listed.

Catalog of metrics for SDLC [34] is a collection of qualitative metrics for all phases of the software development lifecycle. Metrics are based on seemingly random variables present in the development phase. No reasoning for the selection of the metrics is given. Metrics are meant to help detect and assess risks during the phases, but there are no instructions on how to interpret the results.

3.2.3 Security characteristics perspective

The next category is slightly different from the earlier categorizations, which focused on different abstraction levels. This category is based on an interesting idea presented in [21] where the authors examined ten software security metrics based on their security characteristics. The focus of this category is on the features of the security metrics. The following Table 3-3 is combined from the tables presented in [21] and reproduced as such. Feature coverage is marked with a “X” for full feature coverage and a “/” for partial feature coverage.

Table 3-3: Categorization of metrics based on the security characteristics (adapted from [21])

METRIC **CHARACTERISTIC**	Object oriented class design	Security estimation framework	Common Criteria	ISO 27004	Attack surface analysis	CWE	CVSS	CMSS	NIST 800-55	Security metrics for software systems
Authenticity	X		X	X	/	X	X	X	X	/
Confidentiality	X		X	X	/	X	X	X	X	/
Conformance		X	X	X	/	X	X	X	X	/
Detection of attacks	/		/	/	X	X	X	X	/	/
Availability				/		X	X	X	/	/
Integrity	X		X	X	/	X	X	X	X	/
No repudiation			X	/		/	/	/	/	/
Traceability	/	X	X	X		/	/	/	X	/
Conformance (safety)		X	X	X		X	X	X	X	/
Security and health of operator				/		/	/	/	/	
Public health and security				/		/	/	/		/
Commercial damage						/	/	/	/	/
Environmental damage						/	X	/		

Object oriented class design [4], presented more thoroughly in a PhD dissertation titled *Quality Metrics for Assessing Security-Critical Computer Programs* [35], is a method that allows the comparison of two similar object-oriented classes for their security. It uses metrics created from the characteristics of object-oriented classes to analyze the information flow of the program. The dissertation also introduces metrics such as Object-oriented multiclass design, Security metrics for object-oriented programs, and Hierarchical security assessment model. These other metrics can be used together with the object oriented class design metric and are presented in the next section.

Attack surface analysis [36] [37], presented more thoroughly in a PhD dissertation titled *An Attack Surface Metric* [38], is a method of measuring a software system's interaction with its environments. The analysis uses the theory of automata to model the software system and presents methods to measure the attack surface on different programming languages. The analysis results in a quantitative value for the system's attack surface.

CWE [39], the Common Weakness Enumeration, is an attempt at creating a formal common ground for discussing software vulnerabilities and weaknesses. CWE authors upkeep a list or dictionary of common software weaknesses that can occur in the design or implementation of a given software. CWE is designed for educational purposes and is meant for software developers.

CVSS [40], the Common Vulnerability Scoring System, is a method to evaluate individual software vulnerabilities by quantifying the severity of the security issue. The evaluation is platform independent and allows a comparison of vulnerabilities from different vendors.

CMSS [41], the Common Misuse Scoring System, is a sibling metric to CVSS (and a third one called CCSS). CMSS attempts to give quantitative evaluation for software feature misuse vulnerabilities where CVSS targets software flaws and CCSS configuration flaws. The evaluation methods of CMSS and CVSS are similar.

Security metrics for software systems [42] presents a method that uses CVSS scores and the assigned CVEs (Common Vulnerabilities and Exposures, a unified vulnerability ID scheme) as a basis for calculating the level of security of a software package. No reasoning for the formulas used is provided.

3.2.4 Software system perspective

The final category presented in this thesis is based on the idea presented in [43]. In this viewpoint the software system is divided into three levels based on how detailed view they give to the system. This viewpoint is likely to be most familiar to software developers. Table 3-4 shows the abstraction levels used: the system level, design level and the code level. The focus of this category is a technical one, so some previously presented metrics are ruled out from the scope.

Formal analysis for secure architectures [44] is a technique presented in a PhD dissertation, which describes a way to reconstruct the software environment used in constructing the software architecture. The technique allows the comparison of the written architecture against the uncovered architectural assumptions.

Analyzing architectures with Bauhaus [45] [46] is a method for extracting the software architecture from software for threat modeling. The extraction is done with the help of a tool called *Bauhaus*. Security is measured by evaluating whether the discovered architecture exhibits signs for the vulnerabilities discovered during thread modeling.

Measuring security article [17] proposes a method of measuring security by using CVE data and CVSS numbers found on them. The article falls short on providing actual information on working with the data and what kind of results the work would bring.

SEEA to ISO 26262 [47] describes a way to use the Software Error and Effects Analysis (SEEA) [48] in fulfilling the automotive industry standard ISO 26262 requirements for security evaluation of software architecture. SEEA is a process that qualitatively analyzes possible errors in the system components and how the consequences of these errors would propagate throughout the system.

Security metrics for object-oriented programs [35] is a metric for Java based object-oriented programs presented in a PhD dissertation [35]. It combines two object-oriented class based metrics presented in the dissertation with some additional Java specific program code metrics. The aim of the metric is to extend the object-oriented class based metrics into automated tools that could process source code or bytecode for obtaining results.

Table 3-4: Categorization of metrics in software system level concepts.

SYSTEM LEVEL METRICS
Common Criteria
CVSS
CMSS
Security metrics for software systems
Formal analysis for secure architectures
Analyzing architectures with Bauhaus
Measuring security article
SEEA to ISO 26262
Security metrics for object oriented programs
Hierarchical assessment model
DESIGN LEVEL METRICS
Component analysis
Object oriented class design
Object oriented multiclass design
Attack surface analysis
Formal constrain analysis
OCL signature analysis
Risk analysis for security pattern systems
Security pattern analysis
USIE model
Vulnerability index in dynamic architectures
CODE LEVEL METRICS
Attack prone component prediction
ASSM
CCD
Three code metrics

Hierarchical security assessment model [49] [35] is a model that combines three metrics presented in a PhD dissertation [35] to produce a more descriptive quantitative assessment of software. The three metrics this model uses are: object-oriented class design, object-oriented multiclass design, and security metrics for object-oriented programs. The model presents calculation formulas for combining the results from the metrics and gives each program a single quantitative number to describe their security.

Object oriented multiclass design [50] [35] is a metric that allows measurement of security from a multiclass object-oriented design. It is part of the set of metrics presented in a single PhD dissertation [35]. It differs from the single class version in the object oriented design properties that are used to create the metric. The metric provides a way to calculate security quantitatively and gives some guidance in interpretation.

Formal constrain analysis [51] is a process for analyzing the software architecture's compliancy with the security requirements of the software. The process is not described in depth and no evaluation criteria are presented.

OCL signature analysis [52] is a method of describing and analyzing software architectures by using a formal language called Object Constraint Language (OCL). The method does not provide analysis metrics, but instead describes how metrics data can be extracted from the architecture model.

Risk analysis for security pattern systems [53] is a method of performing risk analysis on software that is designed by using security design patterns. The method measures security qualitatively by describing the risk or change in the risk that comes from introducing or removing security design patterns.

Security pattern analysis [19] is another method of analyzing software architecture security by using security design patterns. It is an earlier work made by the author of *Formal analysis for secure architectures* and describes a way to combine the security requirements of software to security design patterns and the metrics associated with the patterns.

USIE model [54], shorthand for User System Interaction Effect model, is a notation framework for describing Service Oriented Architecture diagrams in a way that is more security oriented to allow for improved metrics development and security analysis. The authors of USIE model do not provide the metrics or analysis methods.

Vulnerability index in dynamic architectures [55] is a method for analyzing the dynamic, or runtime, architecture of software for characteristics such as security. The method models the dynamic software architecture as a discrete time Markov chain. The probabilities and calculations of a security related vulnerability index are not defined clearly.

Attack prone component prediction [56] is a model, presented in a PhD dissertation using static code analyzers to demonstrate there is a possibility of determining the components in software that have a higher probability of vulnerabilities. The model first defines the code features that the static analyzers should look for and then describes the connection between vulnerability probability and the static analysis results.

ASSM [57], shorthand for Analyzer-based Software Security Measurement, is a model that combines several code and design level metrics to give a single security indicator value for a software system. The model does not describe how to measure the metrics, but it defines what values to expect from them and how to combine the values into results for the model.

CCD [58] is an acronym given to a set of metrics (in complexity, code churn and developer activity) that define a qualitative method to predict vulnerabilities in individual files. The method first describes the metrics used, and then instructs the user how to calculate and interpret the results. Access to source code management repository is needed in addition to the source code.

Three code metrics [43] is a collection of three source code metrics presented in an article. The article defines the metrics and a calculation formula for a single value, but no instructions for interpreting the results are given.

3.3 Choosing the category and metrics for this study

Upon examining the different categories and their subcategories, it becomes clear that some metrics are unsuitable for the purposes of this thesis. To recap, the focus of this thesis was on software design and how to measure its security. With this in mind, it is time to evaluate the individual categories.

3.3.1 A look at potential categories from this thesis' perspective

The security engineering perspective is a suitable category for process improvement or procurement purposes, but it is not very helpful for software design activities. The subcategory of evaluation is defined to mean external evaluations, which is not the case in this thesis. Therefore, the security engineering perspective and evaluation metrics are ruled out.

The business perspective focuses on economic or management aspects of the security measurement. Because the focus of this thesis is not on the software development lifecycle process, the subcategory of organizational level metrics and the business perspective category are ruled out.

The security characteristics perspective is quite interesting and informative for studying metrics, but it offers very little from the viewpoint of software design. Thus, while the security characteristics perspective is useful for the purposes of the RQ2, it is not practical for discovering what the metrics reveal about the software's design security. We therefore eliminated the security characteristics perspective from our list.

The fourth category -the software system perspective- appears quite suitable for the purposes of this thesis. It has a clear technical focus and a subcategory for the software design. Because the system level or the code level metrics are not really suitable for analyzing software design, the metrics for the case study in the thesis will be chosen from the design level metrics subcategory of the software system perspective.

3.3.2 Additional details of metrics from the chosen category

The design level metrics subcategory of the software system perspective has ten metrics, as seen on Table 3-4. The purpose of this section is to examine the metrics in more detail and find out whether they fulfill the criteria of a 'good' metric given in Section 3.1. The criteria of a good metric was: the metric should be consistently measured, cheap to gather, expressed as a cardinal number and using at least one unit of measure and contextually specific. All metrics not fulfilling the criteria need to be rejected since it is not the goal of this thesis to develop existing metrics further.

The easiest metrics to reject outright are the Formal constrain analysis, the OCL signature analysis, the USIE model and the Vulnerability index in dynamic architectures. Reading the details of these 'metrics' in depth reveals that none actually provide any metrics at all.

The component analysis metric and the Risk analysis for security pattern systems metric share the same trait: they are both qualitative methods. Their analysis is based on qualitative descriptions and evaluations of the security (such as high, low, etc.) and are rejected because the criteria for a good metric calls for quantitative approach for measurement.

There are four metrics left in the design level metrics category: the Object oriented class design, the Object oriented multiclass design, the Attack surface analysis and the Security pattern analysis. The Security pattern analysis is inspected next.

The Security pattern analysis is a qualitative security metric and provides a method for interpreting the results of the metrics. However, it suffers from two flaws. First, the analysis method does not provide the metrics. The metrics used in [19] are only applicable to the specific examples of security design patterns given in the article, and the authors do not give any indication as to how to make more metrics or where to acquire them. Second, the analysis also requires that the inspected software use security design patterns in its design, and is therefore not universally applicable. For these two reasons the Security pattern analysis is rejected.

The Object oriented class design metric, presented in [4], and the Object oriented multiclass design metric, presented in [50], are examined together since they are explained more thoroughly in the same PhD dissertation [35]. Both metrics are consistently measured, since they are based on the properties of object-oriented classes. This makes them relatively cheap to gather, especially on common object-oriented languages, such as Java, which have numerous helpful tools available for extracting this kind of information. The metrics are quantitative and use numbers to express the results. The metrics also use units of measure, such as the number of attributes. Finally, the metrics are related to software design and thus, the object-oriented properties used are generic to object-oriented design. Both metrics are also independent of the programming language used. In conclusion, both of the metrics are good candidates for the case study.

The final metric left in the subcategory is the Attack surface analysis, which is introduced in [38]. The analysis starts by defining the system's attack surface through the theory of automata and then proceeds to define how to measure the attack surface on two different programming languages. The metric can be consistently measured and it is cheap to gather. Additionally, the result of the metric is a number expressing the size of the attack surface and is therefore valid as a unit of measure. Software architecture and design work deal with the environment of the software as well, and since the point of this metric is to measure the exposure of the software to its environments, the metric is context specific. The Attack surface analysis seems to be a good candidate for the study too.

The design level subcategory shrunk from ten to three metrics. However, this does not guarantee the metrics will be useful for the purposes of this thesis. The measurement process of this thesis requires the metric to be implemented after selecting the suitable metric. The next section describes the process of exanimation for the chosen metrics.

3.3.3 Conducting a detailed examination of the chosen metrics

The following two chapters will inspect the three chosen metrics in depth to evaluate their suitability for practical security measurement. Suitability for practical measurement means that there must be a way to implement the metric. The two object-oriented class metrics, introduced by Alshammari in [35], are discussed in Chapter 4 and the attack surface metric, described by Manadhata in [38], is the topic of Chapter 5.

The inspection evaluates whether the metrics fulfill the criteria for the elements of measurement presented in Section 3.1. The evaluation is subjective and qualitative because the source of the elements of measure, [17], only vaguely defines the fulfillment criteria. Table 3-5 repeats the elements of measurement from Section 3.1 and describes the fulfillment criteria of each element. All elements are abbreviated for easier reference in subsequent chapters.

Table 3-5: Five critical elements of measurement (adapted from [17])

ABBR	NAME	DEFINITION
EM1	Property	Identify the property to be measured. Build a model of the phenomenon.
EM2	Metric	Define a metric that quantitatively characterizes the property. Can be a unit of measurement, standard to apply against or scale to evaluate against.
EM3	Measure	Develop a measure, which applies metric to the target. Can be a measuring instrument, formula or other mental device to apply a metric. Should be linear, that is, identical changes in the property value affect the change in measure similarly.
EM4	Measurement	Design the measurement process. Calibration of the measuring device. Collection and availability of the data.
EM5	Value	Each instance of measurement must deliver a result that is composed of a value and an estimate of its accuracy (an error).

In practice the evaluation in Chapters 4 and 5 is achieved by first introducing the theoretical foundation of the metric. After presenting the theoretical foundation, a practical way of applying the theory into reality is presented. During the evaluation, the elements of measurement are discussed when appropriate. Criticism towards the metrics and other remarks of interest are not discussed in Chapters 4 and 5, but are left for later chapters.

Finally, we present ideas and remarks regarding how to perform the practical measurements with current tools. The object-oriented language chosen for these tools is Java because the original authors use Java and because there are a multitude of Java tools available.

4 ANALYSIS OF SECURITY CRITICAL INFORMATION FLOW

As mentioned in the previous chapter, Alshammari introduces four metrics in [35] with the aim of measuring security critical information flow in object-oriented programs. The first section introduces all four metrics and the subsequent sections deal with two metrics chosen for closer inspection.

4.1 Introduction to Alshammari's four metrics

The metrics with their important characteristics are presented in Table 4-1 for easy reference. From now on, Alshammari's metrics will be referenced with the abbreviations from Table 4-1.

Table 4-1: A summary of Alshammari's metrics

ABBR	NAME	TYPE	MATERIAL	NOTE
AM1	Object oriented class designs	Single class metric	Annotated UML class diagram	
AM2	Object oriented multiclass designs	Multiclass metric	Annotated UML class diagrams	Complements 1
AM3	Security metrics for object oriented programs	Full Java program metric	Annotated program source or byte code	Combines 1 and 2 with some extra metrics
AM4	Hierarchical security assessment model	Composition metric	Results from all metrics	Provides formulas for calculating meaning for metrics' results

Alshammari begins his thesis by introducing the two object-oriented design metrics, from which AM1 is meant for single class designs while AM2 is for multiclass designs. Both of these metrics use UML diagrams and can be considered as pure design metrics. He then proceeds to introduce refactoring rules for the UML diagrams and examines their effect on the security of the designs. Discussing the refactoring highlights potential ways to use his metrics' findings in improving the security of a program.

After refactoring, Alshammari proceeds to introduce additional metrics for full Java programs, AM3, which are complementing the object-oriented design metrics he introduced earlier. These new metrics depend on implementing the program and require source code for analysis. [35] It is interesting to note that he also introduces a method for using the previous AM1 and AM2 metrics with Java source code. In addition, his work includes the definition of an automated tool for calculating the values for his metrics.

There appears to be a minor issue in Alshammari's work related to the single class metric AM1 and full program measurement: How is a single class metric calculated when evaluating a program? Alshammari's answer -based on the description of his automated tool in [35]- seems to be to consider all classes as one (i.e. to collect data from all classes to calculate his single class metric).

As a final metric, Alshammari introduces a composite metric AM4, which is meant to help interpret the results from his full Java program metric. Since the Java program metric only complements the earlier metrics, the hierarchical assessment model is actually describing a way of interpreting all of his metrics. The idea in AM4 is that it has four levels of quantitative results that describe the relative security of the program. The results of higher levels are derived from the lower level metrics' results. [35]

The two object-oriented class design metrics have the most solid foundation of Alshammari's metrics. They are language agnostic and do not require any specific implementation level details, but they require special annotations into existing UML diagrams. Both metrics are relative, meaning that they can be only used to compare programs of similar type. [35]

4.2 Theoretical basis: object oriented design properties and data flow

The main idea behind object-oriented class design security metrics is to measure software security from the perspective of potential information flow through a program's object-oriented module structure [35]. The metrics are based on established properties of object-oriented programs and are used in combination with data flow analysis principles that trace potential information flow between high- and low-security system variables [35]. Alshammari describes the goal of his metrics as protecting the confidentiality of the data in regard to the software architecture [35]. Table 4-2 explains the terms Alshammari uses when describing his metrics.

The single class metric AM1 uses two properties of object-oriented classes as the basis: data encapsulation and cohesion. The reason for selecting these properties is not just that they are well known and widely used, but also because their relation to lower level class properties is clearly defined. Furthermore, because the purpose of the AM1 is to measure the information flow through the class, the selected properties align well with two security design principles: “the principle of least privilege” and “reduce attack surface”. [35]

Table 4-2: Terminology used by Alshammari in his thesis (adapted from [35])

TERM	**EXPLANATION**
Classified attribute	*An attribute annotated as “secrecy”*
Instance attribute	*An attribute with separate values for each class instance*
Class attribute	*An attribute with shared value for all class instances*
Classified method	*A method which reads from or writes to a classified attribute**
Unclassified method	*A method which does not interact with classified attributes*
Mutator	*A method that sets the value of an attribute*
Accessor	*A method that reads the value of an attribute*
Classified mutator	*A method that sets the value of a classified attribute*
Classified accessor	*A method that reads the value of a classified attribute*
Critical class	*A class with classified attributes*

* Note: Questions and problems related to indirect access are discussed in Chapters 6 and 8.

Table 4-3: Single class metrics (collected from [35])

	ABBR	NAME	DEFINITION
ACCESSIBILITY	CIDA	Classified Instance Data Accessibility	*The ratio of non-private classified instance attributes to classified attributes in a class.*
	CCDA	Classified Class Data Accessibility	*The ratio of non-private classified class attributes to classified attributes in a class.*
	COA	Classified Operation Accessibility	*The ratio of non-private classified methods to classified methods in a class.*
INTERACTION	CMAI	Classified Mutator Attribute Interactions	*The ratio of mutators that may interact with classified attributes to the maximum possible number of mutators that could interact with classified attributes.*
	CAAI	Classified Accessor Attribute Interactions	*The ratio of accessors that may interact with classified attributes to the maximum possible number of accessors that could have access to classified attributes.*
	CAIW	Classified Attributes Interaction Weight	*The ratio of all methods that could interact with classified attributes to the total number of methods with access to all attributes.*
	CMW	Classified Methods Weight	*The ratio of classified methods to the total number of methods in a given class.*

In other words, the AM1 is divided into two groups: accessibility and interaction metrics. The metrics aim to measure the amount of privilege granted to parts of the program and the relative size of the attack surface of the program. Accessibility metrics try to measure information relevant to the least privileged principle and obtain their information from the data encapsulation properties. Interaction metrics attempt to measure information relevant to the attack surface reduction and draw information from the cohesion properties. [35]

Based on the use of object-oriented class properties and their connection to the security design principles, the AM1 metric fulfills the EM1 criteria. The use of ratios in the AM1 metrics is based on the earlier work by other authors on object oriented quality metrics [35]. Therefore, the EM2 criteria are fulfilled by the definitions of these metrics.

Table 4-3 presents the definitions of the AM1 metrics arranged into the two groups. Table 4-4 presents the rationale behind each AM1 metric. The definitions and rationale are collected from [35].

Table 4-4: Rationale for the single class metrics (collected from [35])

ABBR	RATIONALE
CIDA	*Measures direct accessibility of classified instance attributes. Helps protect classified internal representations of a class from direct access. Higher value means higher accessibility.*
CCDA	*Measures direct accessibility of classified class attributes. Helps protect classified internal representations of a class from direct access. Higher value means higher accessibility*
COA	*Measures potential attack surface size exposed by classified methods. Helps protect classified internal operations of a class from direct access. Higher value means higher surface.*
CMAI	*Measures interactions between mutators and classified attributes in a class. Higher values indicate stronger cohesion between mutators and classified attributes, and thus, more privileges for mutators over classified attributes.*
CAAI	*Measures interactions between accessors and classified attributes in a class. Higher values indicate stronger cohesion between accessors and classified attributes, and thus, more privileges for accessors over classified attributes.*
CAIW	*Measures interactions of classified attributes by all methods of a class. Shows how many potential class interactions are dependent on classified attributes.*
CMW	*Measures the weight of methods in a class that potentially interacts with any classified attributes. Shows the attack surface size based on operations over confidential data.*

The multiclass metric AM2 is designed to complement the single class metric. As the name implies, the metric is no longer aimed at individual classes, but at class structures. The basis for AM2 comes from five quality properties of object-oriented programs: composition, coupling, extensibility, inheritance and design size. The security principles used in the development of the metrics are still the same as AM1 (i.e. "the principle of least privilege" and "reduce attack surface"). Each quality property's contribution to the security of the design is defined separately.

While Table 4-5 shows relations between quality properties and security, the link between the security design principles and quality properties is not as tight as in the single class metric [35]. Regardless, the object-oriented quality properties are well established in the field so the EM1 criteria are fulfilled for AM2 metric.

Table 4-5: Object oriented quality properties' relation to security (collected from [35])

QUALITY PROPERTY	RELATION TO SECURITY
Composition	*Composition means a lifetime dependency between an object (outer class) and its composite objects (inner classes). Assumption: inner classes are only accessible by outer classes. Usage of inner classes thus increases security.*
Coupling	*Coupling means the degree of interaction an object has with other objects. Objects with high coupling are greater target for successful attacks than objects with small coupling.*
Extensibility	*Extensibility means that a class or method can be extended by other classes or methods. Extensibility is considered to be bad for security and should be discouraged unless considered necessary.*
Inheritance	*Inheritance allows to provide classes with generalizations and special relationships. Inheritance allows reuse. Inheritance could allow subclasses access to classified information.*
Design size	*Measures classes in a design. Design size has a large impact on functionality and reusability.*

To put it differently, the AM2 metrics are divided into five groups according to the chosen quality properties of object oriented programs. The metrics aim to measure the amount of privilege granted to the parts of the program and the relative size of the attack surface of the program, as was the case with the single class metric. Connection of the metric groups to the security design principles is loose, but each individual metric is connected to either principle. [35]

Table 4-6 presents the definitions of the AM2 metrics arranged into the object oriented quality properties. Table 4-7 presents the rationale behind each AM2 metric. The definitions and rationale are collected from [35]. As with AM1 metric, the definitions of AM2 metrics fulfil the criteria for EM2.

Table 4-6: Multiclass metrics (collected from [35])

	ABBR	**NAME**	**DEFINITION**
COMPO-SITION	CPCC	Composite-Part Critical Classes	*The ratio of critical composed-part classes to the total number of critical classes in a design.*
COU-PLING	CCC	Critical Classes Coupling	*The ratio of all class links with classified attributes to the total number of possible links with classified attributes in a given design.*
EXTENSI-BILITY	CCE	Critical Classes Ex-tensibility	*The ratio of the non-finalized critical classes in a design to the total number of critical classes in that design.*
	CME	Classified Methods Extensibility	*The ratio of the non-finalized classified methods in a de-sign to the total number of classified methods in that de-sign.*
INHER-ITANCE	CSP	Critical Super-classes Proportion	*The ratio of critical superclasses to the total number of critical classes in an inheritance hierarchy.*
	CSI	Critical Super-classes Inheritance	*The ratio of the sum of classes which inherit from each critical superclass to possible inheritances from all critical classes in a class hierarchy.*
	CMI	Classified Methods Inheritance	*The ratio of classified methods which can be inherited in a hierarchy to the total number of classified methods in that hierarchy.*
	CAI	Classified Attrib-utes Inheritance	*The ratio of classified attributes which can be inherited in a hierarchy to the total number of classified attributes in that hierarchy.*
DESIGN SIZE	CDP	Critical Design Pro-portion	*The ratio of critical classes to the total number of classes in a design.*

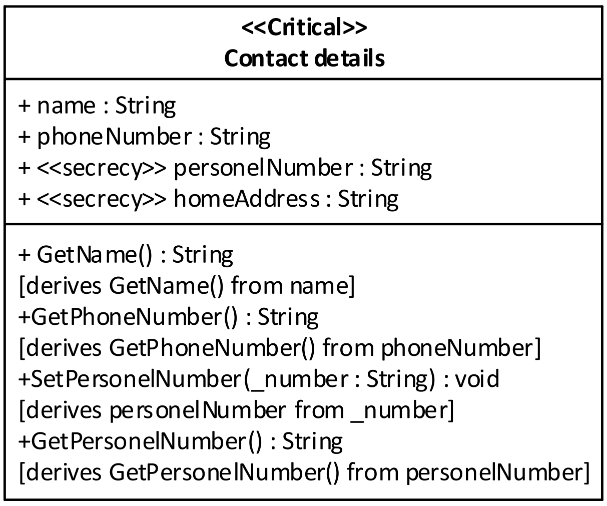
Table 4-7: Rationale for the multiclass metrics (collected from [35])

ABBR	PRINCIPLE	RATIONALE
CPCC	attack surface	*Measures the inner and outer class structure of critical classes. Aims to reward the use of inner classes to hold classified data. Higher values indicate higher numbers of outer classes with classified data.*
CCC	privilege	*Measures the degree of security relevant coupling between classes and classified attributes. Aims to penalize programs with high coupling. A high value indicates a high degree of coupling.*
CCE	attack surface	*Measures the extensibility of critical classes. Aims to penalize designs with extensible critical classes. Higher value means higher extensible critical classes.*
CME	attack surface	*Measures the proportion of non-finalized classified methods to all classified methods. Aims to reward designs with inextensible classified methods. High value means high amount of extensible classified methods.*
CSP	attack surface	*Measures the proportion of critical superclasses to all critical classes in inheritance hierarchy. Aims to penalize the use of critical superclasses. High value means a high amount of critical superclasses.*
CSI	privilege	*Measures the ratio of inheritance from critical superclasses versus all critical classes. Penalizes class hierarchies where critical classes appear near the top. High value means high amount of classes can inherit from critical superclasses.*
CMI	attack surface	*Measures the proportion of classified methods that are exposed to inheritance. Penalizes the use of inheritance for classified methods. High value means high amounts of classified methods that can be inherited.*
CAI	attack surface	*Measures the proportion of classified attributes which are exposed to inheritance. Penalizes the use of inheritance for classified attributes. High value means a high amount of classified methods that can be inherited.*
CDP	attack surface	*Measures the proportion of critical classes to all classes. High value means a high amount of critical classes in the design compared to other designs of same size.*

4.3 Practical method for single class analysis

The source material for single class metrics are the UML class diagrams, although the standard UML notation is not enough. All UML diagrams need to be annotated with UMLsec [59] and SPARK programming language [60] specific annotations, even though only a handful of these specific annotations are used in the measurement process. [35]

From the UMLsec, we use the labels “secrecy” and “critical”. The label of “secrecy” is assigned to any data attribute that needs to remain confidential. The label of “critical” is assigned to any class that holds attributes labeled as “secrecy”. Figure 4-1 shows an example of the annotations used in the metric. The decision of what data is actually confidential is left for the user of the metric to decide. This means that the accuracy of the user’s annotations has a high impact on the accuracy of the metrics. [35]



<<Critical>>
Contact details
+ name : String
+ phoneNumber : String
+ <<secrecy>> personelNumber : String
+ <<secrecy>> homeAddress : String
+ GetName() : String
[derives GetName() from name]
+GetPhoneNumber() : String
[derives GetPhoneNumber() from phoneNumber]
+SetPersonelNumber(_number : String) : void
[derives personelNumber from _number]
+GetPersonelNumber() : String
[derives GetPersonelNumber() from personelNumber]

Figure 4-1: Example of the metric’s annotations (adapted from [35])

From the SPARK programming language only the subroutine data flow annotation is used, wherein the user of the metric describes the possible data flow between variables and parameters. The idea is to highlight values of a specific variable that might be derived from the value of another variable elsewhere in the system. [35]

Table 4-8 shows the exact formulas for counting metrics. The results of the metrics are scaled to the range 0 to 1 [35]. Since the AM1 metric is counted with a formula, it fulfils the criteria of EM3.

A simple way to interpret the results of the calculations is to present them in a radar chart such as that found in Figure 4-2, derived from [35]. The closer to zero (the center of Figure 4-2) the value is, the better is the result. Values between designs are usually not uniformly better or worse, so a decision between the designs must be made using knowledge of each metric’s denominator. [35]

Table 4-8: Single class metrics' formulas (collected from [35])

ABBR	FORMULA	EXPLANATION
CIDA	$\frac{number\ of\ NCIA}{number\ of\ CA}$	NCIA = non-private classified instance attributes CA = classified attributes
CCDA	$\frac{number\ of\ NCCA}{number\ of\ CA}$	NCCA = non-private classified class attributes CA = classified attributes

COA	$\frac{number\ of\ NCM}{number\ of\ CM}$	NCM = non-private classified methods
		CM = classified methods

CMAI	$\frac{\sum_1^{number\ of\ CA} number\ of\ mCA}{number\ of\ MM\ \times number\ of\ CA}$	CA = classified attributes
		mCA = mutator methods that can access classified attributes
		MM = mutator methods

CAAI	$\frac{\sum_1^{number\ of\ CA} number\ of\ aCA}{number\ of\ AM\ \times number\ of\ CA}$	CA = classified attributes
		aCA = accessor methods that can access classified attributes
		AM = accessor methods

CAIW	$\frac{\sum_1^{number\ of\ CA} number\ of\ nCA}{\sum_1^{number\ of\ A} number\ of\ nA}$	CA = classified attributes
		nCA = methods that can access classified attributes
		A = attributes
		nA = methods that can access attributes

CMW	$\frac{number\ of\ CM}{number\ of\ M}$	CM = classified methods
		M = methods

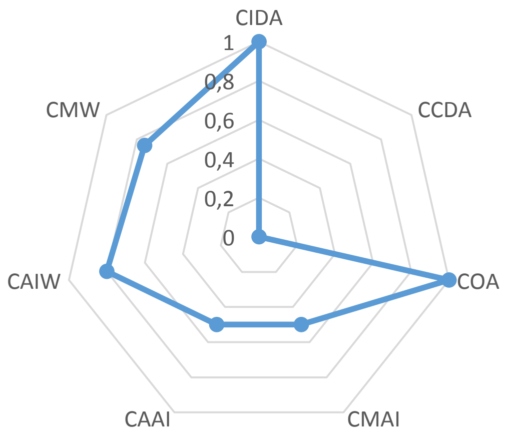


Figure 4-2: Radar chart from single class metrics with example data [35]

4.4 Practical method for multiclass design analysis

The practicalities of multiclass metrics are exactly the same as for single class metrics, but naturally the UML class diagram is larger.

Table 4-9 shows formulas for multiclass metrics and Figure 4-3 shows an example of a radar chart as found in [35]. As with the AM1 metric, the AM2 metric is counted with a formula and therefore fulfils the criteria for EM3.

Table 4-9: Multiclass metrics' formulas (collected from [35])

ABBR	FORMULA	EXPLANATION

CPCC	$1 - \frac{number\ of\ CP}{number\ of\ CC}$	CP = composed-part critical classes
		CC = critical classes

CCC	$\frac{\sum_{1}^{number\ of\ CA} number\ of\ aCA}{(number\ of\ C - 1)\ \times number\ of\ CA}$	CA = classified attributes
		aCA = classes, which interact with classified attributes
		C = classes

CCE	$\frac{number\ of\ ECC}{number\ of\ CC}$	ECC = extensible critical classes
		CC = critical classes

CME	$\frac{number\ of\ ECM}{number\ of\ CM}$	ECM = extensible classified meth-ods
		CM = classified methods

CSP	$\frac{number\ of\ CSC}{number\ of\ CC}$	CSC = critical superclasses
		CC = critical classes

CSI	$\frac{\sum_{1}^{number\ of\ CSC} number\ of\ nCSC}{(number\ of\ C - 1)\ \times number\ of\ CC}$	CSC = critical superclasses
		nCSC = classes, which may inherit from the critical superclass
		C = classes
		CC = critical classes

CMI	$\frac{number\ of\ MI}{number\ of\ CM}$	MI = classified methods that may be inherited CM = classified methods
CAI	$\frac{number\ of\ AI}{number\ of\ CA}$	AI = classified attributes that may be inherited CA = classified attributes

CDP	$\frac{number\ of\ CC}{number\ of\ C}$	CC = critical classes
		C = classes

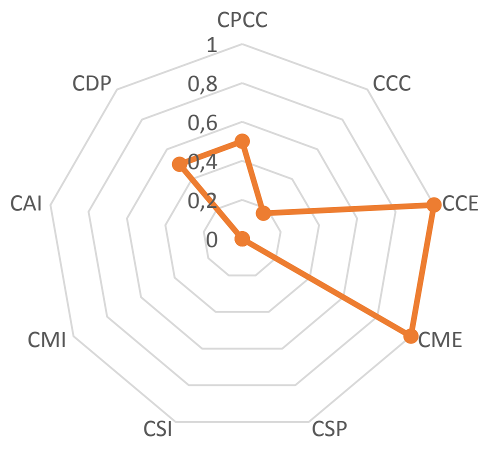


Figure 4-3: Radar chart from multiclass metrics with example data [35]

4.5 Object oriented design metrics: In practice

AM1 and AM2 metrics require UML diagrams with UMLsec and SPARK annotations, however, no premade tool comes with support for both of these and SPARK is not commonly used with UML [61]. It is a subset of ADA programming language [62] and the tools are not meant for processing UML diagrams.

A quick search on the Internet through the major UML tool vendor's products reveals that UMLsec is not a common UML extension either. UMLsec seems to receive greatest support from the UMLsec author's own tools [63] [64], but these are not compatible with the AM1 or AM2 metrics.

The AM3 metric, which encompasses AM1 and AM2 metrics, does not use the UMLsec or SPARK syntaxes anymore. AM3 uses Java source code annotation to mark confidential data. The data flow analysis is based on a type inference model that Alshammari presents and it is also done on the source code. The reason for using this type inference model seems to be the need for creating rules for an automated source code analyzer [35] [65], but use of the inference model does not seem to be necessary for the application of the AM1 and AM2 metrics. Alshammari's own automated source code analyzer is not available at the writing of this thesis and attempts to contact Alshammari have been unsuccessful.

In conclusion, it seems that the original way of using the AM1 and AM2 metrics are not feasible for existing software projects. Most projects have source code and binaries available, but are not annotated in any special way. UML diagrams and architecture documentation do not usually exist for small open source software projects, which would be a good target of evaluation for this thesis. Building an automated tool is out of the scope of this thesis because of the research questions and due to the time and effort required to build a Java bytecode analyzer, annotation processor, and logic for data flow analysis.

The AM3 metric provides an idea for practical application because it uses Java source code for the analysis. The AM1 and AM2 metrics can be applied to small to medium size software projects manually. By looking at the Table 4-8 from Section 4.3 and Table 4-9 from Section 4.4, it is quite clear that most of the metrics' data can be gathered with only a small effort.

So the steps for applying the AM1 and AM2 metrics in practice are:

1) Locate the relevant Java source code files.
2) Decide and locate the classified data attributes.
3) Identify other relevant data, such as classes, attributes and methods.
4) Gather easy-to-locate data manually.
5) Gather additional data that is difficult to locate using text parsing tools (such as grep) or IDE (such as Eclipse or IntelliJ IDEA) search functions.

The criteria for EM4 is fulfilled by including a measurement process for the metrics. Using manual work naturally does bring an extra source of error to the calculations, but the results of this process can be easily verified due to the simplicity of the required data. Since the biggest sources of error (i.e. the decision of confidential data and the manual work) are known, and the process always delivers results, the criteria for EM5 is fulfilled.

Both AM1 and AM2 metrics pass the criteria for the critical elements of measurement, and can therefore be used in the case study of this thesis.

5 ANALYSIS OF THE SYSTEM ATTACK SURFACE

The analysis of a system’s attack surface is a metric from Manadhata’s PhD dissertation [38] and is based on the idea that a system’s potential security can be measured by the size of the system’s attack surface. An attack surface is defined as: the ways in which an attacker can enter the system and potentially cause damage. [38]

Manadhata’s metric does not fit well into the software system categorization from Section 3.2.4 because the metric includes parts of all three categories. However, it mostly deals with concepts that are related to software design or software architecture, so it is justified to include this metric into this thesis. No further explanations are given at this point, since the next section explains the basic concepts behind the metric, shedding light the issue.

The attack surface metric is a relative metric that only allows comparison of similar systems and does not measure code quality. Manadhata states that the metric “measures the potential of being more insecure than the other”. The attack surface metric is not dependent on any programming language or a specific implementation. In theory, it could use design phase artifacts, but in practice the metric requires source code and binaries of the program. [38]

5.1 Theoretical basis: Software system as an I/O automata model

The basic idea of an attack surface is quite intuitive, consisting of three different elements [38]:

1) *Channels* that the attacker uses to connect to the system.
2) *Methods* that the attacker uses on the system.
3) *Data items* that the attacker sends or receives from the system.

All three elements are referred together as *resources* [38], which are present in the system and its environments. However, not all resources contribute equally to the system’s attack surface, leading to the steps involved in the attack surface calculation. First, the resources of the attack surface must be found with the entry- and exit-point framework. Then, their potential to cause harm must be estimated, and finally, the effort required from the attacker must be considered. The logic is intuitive: easily reached high damage resources contribute more to the attack surface than hard to reach resources of the same kind. The damage refers only to technical damage in this context and does not consider business aspects. [38]

The software systems are modeled as I/O automata in attack surface metric. In fact, all other actors such as the other software systems, users, data storage etc., are considered I/O automata. The channels of a system are modeled as state variables of the automaton and the methods of a system are modeled as actions of the automaton. Entry points are methods in the system that receive data from the environment, while exit points are methods that send data to the environment. Entry and exit points can be direct or indirect, but the indirect entry points are not discussed any further due to problems with finding them automatically. [38]

The I/O automata model is not presented in more detail here, since it is not necessary for understanding the content of this thesis, nor for the application of the attack surface metric. Nevertheless, Manadhata proves mathematically that the attack surface calculations and some of its features are valid. Manadhata's model of I/O automata fulfills the criteria for EM1 and the calculations and features he provides fulfill the criteria for EM2.

The attack surface metric calculation has three phases [38]:

1) Identification of entry and exit points (methods), channels and untrusted data items.
2) Estimation of damage potential-effort ratio for each of the resources.
3) Calculation of the attack surface from the results of step 1 and 2.

The term 'untrusted data item' refers to data items that the entry points read or exit points write. However, in the calculations, untrusted data items refer to data coming from data storage in the environment as opposed to data that is sent directly to entry points by an attacker. Data sent by an attacker directly is already counted in the calculations for the methods. [38]

The result of the attack surface metric is a quantified triple that has values of the sums of damage potential-effort ratios in the order of: methods, channels and untrusted data. The result is not dependent on the attacker. Only the system design and the inherent properties of the system have an effect on the results. [38]

5.2 Practical method for analyzing a Java program attack surface

Manadhata's theoretical model is too abstract for any real world usage, so he introduces two practical methods of his attack surface metric: one for C language and one for Java. The methods share similarities, but are different, especially the identification of entry and exit points [38]. The Java program analysis method is designed to measure programs on a Java application server platform (SAP NetWeaver), so the analysis is likely to be less optimal or have issues outside such an environment.

Table 5-1: Overview of the attack surface measurement process

PHASE	STEPS	RESULTS
1 (Identification)	I. Find methods.	*Number of methods.*
	II. Find channels.	*Number of channels.*
	III. Find untrusted data items.	*Number of untrusted data items.*
2 (Damage potential-effort)	IV. Estimate potentials.	*Numeric amounts in potential groups.*
	V. Organize potentials.	*Potential groups in descending order.*
	VI. Estimate effort.	*Efforts for the potential groups.*
	VII. Organize effort.	*Efforts in descending order.*
	VIII. Assign numeric values.	*Coefficients for ratio calculations.*
3 (Calculation)	IX. Calculate method sums.	*Sum of all methods' damage potential-effort ratios.*
	X. Calculate channel sums.	*Sum of all channels' damage potential-effort ratios.*
	XI. Calculate data item sums.	*Sum of all untrusted data items' damage potential-effort ratios.*
	XII. Present results.	*Attack surface triple.*

The practical method requires access to the source code and the binaries of the Java program. Table 5-1 gives an overview to the measurement process and the steps [38]. Table

5-2 describes the detailed steps taken in the first phase of the calculation where the resources taking part in the attack surface are identified. The details of the calculation are collected from [38]. The results from this phase are numeric amounts for each resource.

Details of the second phase, where the damage potential-effort is estimated, are presented in Table 5-3. A key point in phase two is the assignment of numeric values or the step between the values being highly subjective and completely dependent on the user of the metric. The numeric values should be decided based on the knowledge of the system and its environments. [38]

Table 5-2: Identification of attack surface resources (collected from [38])

STEP	DETAILS
IA Find direct entry points	Direct entry point (a method that receives data) is one of the following: 1) A method that is in the public interface and receives data as input 2) A method that invokes another system's interface method and receives data as result 3) A method that invokes Java I/O library read methods.
IB Find direct exit points	Direct exit point (a method that sends data) is one of the following: 1) A method that is in the public interface and sends data as result 2) A method that invokes another system's interface method and sends data as input 3) A method that invokes Java I/O library write methods.
II Find channels	Monitor the runtime behavior of a system: 1) Identify channels opened by the system. 2) Determine protocol and access rights level for each detected channel.
III Find untrusted data items	Monitor the runtime behavior of a system: 1) Identify untrusted data items accessed by the system. 2) Determine the type and access rights level of each untrusted data item.

Table 5-3: Estimation of damage potential-effort ratio (collected from [38])

STEP	DETAILS
IV Estimate potentials (privileges)	In Java, the method's data sources or destinations are used in grouping the potentials. The method belongs to one of the following groups: 1) Method receives or sends data as input parameter. 2) Method receives or sends data to external data store. 3) Method receives or sends data to other systems in the environment. For channels, the channel type (for example the protocol) is used in the grouping. For untrusted data items, the data item type (for example a file) is used in the grouping.
V Organize potentials (privileges)	Organize the potential groups in descending order from highest to lowest potential.
VI Estimate efforts (access rights)	In Java, the method's access rights level is used in grouping the efforts. The method belongs to one of the following groups: 1) Method is in public interface. 2) Method is in internal interface. For channels, the channel access right (for example "remote unauthenticated") is used in the grouping. For untrusted data items, the data item access right (for example the username) is used in the grouping.
VII Organize efforts (access rights)	Organize the effort groups in descending order from highest to lowest effort.
VIII Assign numeric values	In numeric value assignment all potentials and efforts must get a value. The value itself is subjectively decided. For example, the group in the first place of the ordering receives the amount of groups as the numeric value, the second one receives the amount minus one, etc.

The third phase of the metric is the simplest of them all. This phase includes only the calculation of the sums of the damage potential-effort ratios (abbreviated as DER) and presenting them as the attack surface metric triple. Table 5-4 describes the third phase with which the criteria for EM3 is fulfilled. Note that even though Table 5-1 describes the measurement process, it does not fulfill the criteria for EM4 because it lacks the discussion of data collection and availability.

Table 5-4: Calculations for attack surface value (collected from [38])

STEP	DETAILS
IX Calculate method sums	The DER sums for methods are calculated with the formula: $\sum_{1}^{number\ of\ groups} number\ of\ methods \times \frac{potential\ (privilege)}{effort\ (access\ right)}$
X Calculate channel sums	The DER sums for channels are calculated with the formula: $\sum_{1}^{number\ of\ groups} number\ of\ channels \times \frac{potential\ (privilege)}{effort\ (access\ right)}$
XI Calculate data item sums	The DER sums for untrusted data item are calculated with the formula: $\sum_{1}^{number\ of\ groups} number\ of\ data\ items \times \frac{potential\ (privilege)}{effort\ (access\ right)}$
XII Present results	The attack surface value is the triple: (STEP IX results, STEP X results, STEP XI results)

Even though the attack surface metric does not provide many numeric values, it can be visualized for easier interpretation and comparison of systems. Figure 5-1 shows an example of a bar chart as found in [38].

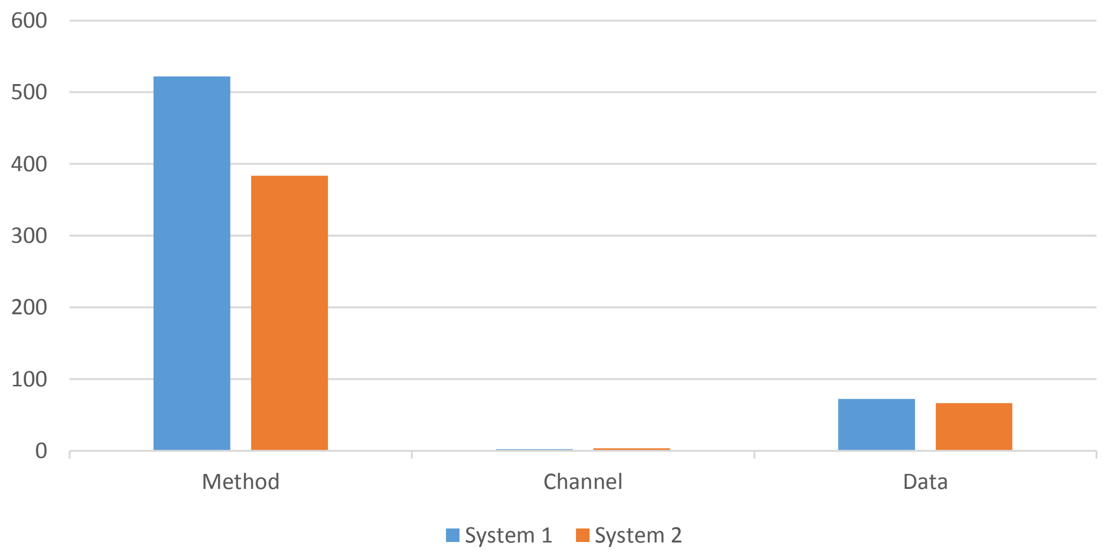


Figure 5-1: Bar chart of attack surface metric from example data [38]

5.3 Using the attack surface analysis in practice

As mentioned previously, applying the Attack surface metric to Java software is different from applying it to a C-program. Manadhata uses SAP Netweaver platform in his application. SAP Netweaver is a Java application server, among many other things, and similar enough to a Java Virtual Machine (JVM) for the purposes of this thesis. This means that the application of the Attack surface metric on Java programs can be repeated on normal computer installations running JVM.

The attack surface analysis requires access to both source code and the compiled binary code of the software. In theory, the whole analysis could be completed with only the compiled binary, but some of the required data for the metric is easier to obtain manually from the source code. In fact, there are no automated tools for performing the attack surface analysis, requiring a manual analysis with the aid of certain tools that are presented subsequently.

The data collection for the method calculations (entry and exit points) requires the examination of method calls and groups them into three categories. The challenge here is that typical Java debug tools are geared towards helping developers find issues, rather than specific information required by the metric. A tool called Java Call Graph Utilities that helps find all of the method calls *from* the library is introduced in [66]. This tool requires compiled code and the results require manual examination to find the relevant calls for this metric. The Call Graph Utilities do not help with some categories of the methods that can be called that require manual examination of the source code.

Data collection for the channel calculations can be completed using a Microsoft tool called The Process Monitor [67]. A single Java application running inside the JVM process can be monitored for all network connections, and the requisite data is captured easily.

The data collection for the untrusted data item calculations can be also completed using a Microsoft tool. This time the tool is named Process Explorer [68]. As with the channel data, the required data is easily acquired by monitoring the JVM process.

Since the phases of the measurement process were explained in the previous section in Table 5-1 alongside the data collection explanation, the criteria is fulfilled for EM4. Sources of error for this metric are the manual assignment of numeral values and the manual work. Because each instance of measurement produces a value, the criteria for EM5 is fulfilled.

The Attack surface metric passes the criteria for the critical elements of measurement, so it can be used in the case study in this thesis.

6 THE PRACTICAL ANALYSIS

This chapter introduces the practical part of this thesis. First a brief look at the theoretical background used in the case study research of this thesis, followed by an introduction of the case study target. The final part of the chapter explains the execution of the case study analysis. The case study is used for studying RQ3 (what metrics reveal about security) in this thesis, while a literature review was used to study RQ1 and RQ2 in the earlier chapters.

6.1 Brief theoretical background for the case study

A case study is defined in [69] as an empirical enquiry that uses multiple sources to study one instance of a contemporary software engineering phenomenon in its context. By examining RQ3, it becomes clear that the case study is both exploratory and descriptive. An exploratory case study is “finding out what is happening, seeking new insights, and generating ideas and hypotheses for new research” [69]. A descriptive case study describes the current status of the phenomenon [69].

The data used in this case study is both quantitative (derived from the metrics) and qualitative (used to interpret the results of the metrics). According to [69], such “mixed methods” data studies often provide better understanding of the phenomenon in question. However, this case study does not provide conclusions that would be of statistical significance [69].

Multiple possible sources of error in this case study exist despite seemingly quantitative metrics, primarily because the metrics have points requiring subjective decisions on the numeric values. Applying the metrics manually might be error prone and vulnerable to subjective bias while analyzing the source material. Triangulation can be used in a case study to reduce the effect of errors [69]. There are three kinds of triangulation types used in this study: data, methodological and theory triangulation. Data triangulation means “using more than one data source or collecting the same data at different occasions” [69]. In this case study, multiple versions of the target software are used. Methodological triangulation means: “combining different types of data collection methods” [69]. This case study uses qualitative interpretation of the quantitative results provided by the metrics.

Theory triangulation means “using alternative theories or viewpoints” [69]. There are two different metrics used in this case study.

There are five steps in case studies [69] that can be mapped to parts of this thesis. The following list (from [69]) illustrates the phases and how they correspond to the thesis:

1) *Case study design*. Chapters 1 and 6.
2) *Preparation for data collection*. Sections 4.5 and 5.3.
3) *Collecting evidence*. Collected data presented on Section 6.3 and Chapter 7.
4) *Analysis of collected data*. Chapter 7.
5) *Reporting*. Chapters 6, 7 and 8.

The theory of case studies is not explained in further detail due to the small size of the case study in this thesis.

6.2 The case study target: Apache James mail server

The Apache James, shorthand for Java Apache Mail Enterprise Server [70] is a Java-based email and news server [71]. James was given an independent project status at Apache around 2003, although the James project had started previously [71]. The goal of the Apache James project was to build a server that provided support for multiple different email and news protocols [71] [72]. The development of James came to a halt around 2012 [73].

James is built using Apache Avalon project [71], a web application framework [74] that Apache worked on during 1999-2004 [75]. The Apache Avalon project evolved from a project that gave Java servlet support to the Apache HTTP server. [74] The Avalon project was focused on the server side components [74]. After the Apache Avalon project closure, a conversion to Spring web application framework was planned [73]. The conversion plans seem to originate at least from the year 2007 [73].

James has a modular structure, where the server and other parts delivering different functionalities are separated [76]. One of the key features advertised in James is the Mailet API, which allows the creation of mail processing applications (such as aliasing, forwarding, etc.) [71]. Due to the modular structure and the APIs offered, the developers advertise James as “the complete email application platform” [76].

As mentioned earlier, the server development seems to have stopped in 2012 [73]. In fact, most major development in James seems to have occurred during 2002-2004, which saw the addition of support for major protocols such as NNTP and SMTP [Appendix B]. James version 3 was supposed to add IMAP protocol support, but version 3 has been under development since 2003 and is currently in beta [73].

Because major development of James seems to have stopped in 2012, one may ask why it was chosen for the case study. Firstly, James is built with Java, so the analysis can be repeated on multiple platforms. Both metrics also used Java as their chosen application language. Second, James is an email server and therefore uses network resources and is considered a high value target for malicious users. Finally, Alshammari uses James in his thesis, providing comparison data [35, pp. 139-143] for this study.

James also has useful attributes for the analysis portion of the case study because of three known vulnerabilities in James resulting from the developers forgetting to sanitize the inputs and check the error conditions. Despite many efforts to find all James related vulnerabilities and their source, it seems that the general interest in James has diminished as the development has stopped. Table 6-1 provides a brief look at the vulnerabilities and Appendix A presents their details.

The versions of James used in this case study are those that can be downloaded from the Apache download repository at [77], which means versions from 2.1.0 to 2.3.2.1. A practical note for accessing the series binaries of James version 2: James uses the Avalon framework, which means that the binaries are packed twice in JAR packages. Unpacking this double JAR package (called a SAR package) with normal tools reveals a normal executable JAR.

Table 6-1: Summary of known vulnerabilities of James

ID	TYPE	FIXED IN	COMPONENT AFFECTED
CVE-2004-2650	Denial of service	2.2.0	Spooler functionality
CVE-2006-2806	Denial of service	2.3.0	SMTP processing
CVE-2015-7611	Remote command execution	2.3.2.1	File based user repository

6.3 Executing the case study analysis

This section presents the details of how the case study research was conducted and the intermediate results that were acquired during the analysis. This section discusses general remarks regarding practicalities, followed by analyses of metrics in order of single class, multiclass and attack surface.

6.3.1 General remarks about the analysis

The codebase analysis of James was only performed on the subfolder "\src\java\org\apache\james" in the James source package, which contains the Java source code for James server components. The analysis was conducted using several small Windows utilities as follows:

- WinMerge [78], to analyze the differences between the versions.
- GrepWin [79], to search for the necessary text strings within the codebase.
- Java Call Graph tool [66], to list the function calls from the James binary.
- Sysinternals Suite tools [67] [68], to monitor James runtime behavior.

The following sections will summarize the analyses, including practical remarks when deemed useful for the study. The practical remarks are any considerations that needed to be accounted for when apply the metrics to a target.

The intermediate results presented are those that were used to calculate the metrics' results in Chapter 7. Not all individual intermediate results are presented though, as they provide little value for the bigger picture. The analysis took approximately 40 hours of work, including planning of the analysis, conducting the actual analysis and resolving any encountered issues.

6.3.2 Analysis using single class metrics

The starting point of the analysis was the James 2.1.0 version, from which, all of the values were calculated thoroughly. Next, differences between versions 2.1.0 and 2.1.1 were inspected and the values were altered accordingly. The analysis continued in this fashion until version 2.3.2.1. Table 6-2 recollects the terms used in the Single class metric equations, while equations and terminology are the same as in Table 4-8.

Table 6-2: Single class OO metrics’ equation terms

TERM	EXPLANATION
NCIA	*Non-private classified instance attributes (attributes with individual values to all classes)*
CA	*Classified attributes (attributes marked as confidential data)*
NCCA	*Non-private classified class attributes (attributes with single value to all classes)*
NCM	*Non-private classified methods*
CM	*Classified methods (methods that access classified attributes)*
mCA	*Amount of the mutator methods that can access classified attributes*
MM	*Mutator methods (methods that change attribute values)*
aCA	*Amount of the accessor methods that can access classified attributes*
AM	*Accessor methods (methods that read attribute values)*
nCA	*Amount of the methods that can access classified attributes*
nA	*Amount of the methods that can access attributes*
M	*Methods*

The Classified Attributes in James are the same as used by Alshammari. [35, p. 139] All versions have three in class DefaultUser: username, hashedPassword and algorithm. In addition from version 2.2.0 onwards, there are two more in class Account: fieldPassword and fieldUser.

Before starting the single class metric analysis, several decisions were necessary due to ambiguities. Alshammari does not explain how the single class metrics are supposed to work with complete projects, but from his results [35, p. 140] it is clear that he is applying them to projects. The assumption taken here is that the single class metric calculation includes all classes in the project. This means that not only is the class with classified attributes counted, but all classes. Needless to say, this decision has major impact on the metric results.

Alshammari also fails to explain how abstract or interface classes are considered in Java language, so in this thesis both of the class types are counted normally in the metrics. The reason for this decision is: interface classes can have classified attributes in their method signatures whereas abstract classes have already implemented methods that can access classified attributes. Interface and abstract classes were as far as object-oriented specialties were considered. Class inheritance and the effect of inheritance to the calculations was not considered in this thesis.

The final problem at the onset of analyses was the lack of an exact definition for 'private'. In Alshammari's work there are only two access right levels: private and non-private [35], whereas the Java language has four access right levels. This issue is not clarified in Alshammari's work, so in this thesis Java access right "private" corresponds to metric access right "private" and all of the other Java access rights correspond to metric access right "non-private".

Table 6-3 presents the intermediate calculation results during the case study and some notes/remarks related to the calculation of the intermediate results follow.

Table 6-3: Single class OO metrics intermediate calculation results

VERSION	**NCIA**	**CA**	**NCCA**	**NCM**	**CM**	**mCA**	**MM**	**aCA**	**AM**	**nCA**	**nA**	**M**
2.1.0	0	3	0	27	28	1	14	29	41	30	1171	1091
2.1.1	0	3	0	27	28	1	14	29	41	30	1175	1095
2.1.2	0	3	0	27	28	1	14	29	41	30	1177	1095
2.1.3	0	3	0	27	28	1	14	29	41	30	1191	1104
2.2.0	0	5	0	33	34	3	23	33	95	36	1479	1601
2.3.0	0	5	0	36	37	3	23	39	98	39	1695	1973
2.3.1	0	5	0	36	37	3	23	39	98	39	1695	1973
2.3.2	0	5	0	36	37	3	23	39	98	39	1695	1974
2.3.2.1	0	5	0	37	38	3	23	40	99	40	1696	1979

The Classified Method (CM) count includes methods from the class that contains Classified Attributes (CA), which were called ‘critical classes’. The CM count also includes some classes that indirectly use CAs. Classes counted using CAs indirectly were classes implementing the User interface, which is also implemented by the DefaultUser class.

In broader terms, Alshammari is a bit unclear on how to deal with indirect references. His text recognizes indirect referencing from time to time, but does not deal with the subject in a coherent fashion [35]. This is especially problematic when defining concepts such as the Classified Method.

The Mutator Method (MM) and the Accessor Method (AM) calculations use the phrase “could potentially interact” (changed to “which may access” in Table 6-2) in some of their formulas. Alshammari does not define exactly what he means with this potential interaction. He only mentions that the MMs (and AMs) need to be in the scope of CAs for this calculation, which means that only classes with CMs are considered. In this thesis “may access” or “could interact” is defined as all methods in the scope that change (or access, in case of the AMs) attributes.

The AMs also have an interesting problem with indirect referencing. The CAs are stored in the class DefaultUser and are often used through the interface User. This behavior is detected by the metrics. However, sometimes a mailbox is fetched using the username (which is a CA) in a text string instead of a class attribute. An example of this behavior is in file James.java at method getUserInbox. Using a text string in similar fashion to a CA is unnoticed in the metrics since it does not use the classes DefaultUser or User.

Calculating all methods that can access any attributes proved quite difficult, but were estimated by calculating the number of distinct attributes a method uses (not counting internal method variables of course). Method amount calculations did not have an exact scope defined, so all interfaces and abstract methods were counted too.

Now it is possible to calculate results for the single class OO metrics with the above considerations and with the intermediate results from the Table 6-3.

6.3.3 Analysis using the multiclass metrics

The intermediate calculation results of the multiclass OO metrics are presented in a similar way to the single class OO metrics. There are fewer remarks in this section, since the multiclass metrics are much simpler to collect relative to the single class metrics. Table 6-4 explains the equation terms used in the intermediate results.

Table 6-4: Multiclass OO metrics' equation terms

TERM	**EXPLANATION**
CP	*Composed-part critical classes (subclasses with CA)*
CC	*Critical classes (classes with CA)*
cCA	*Amount of the classes, which interact with classified attributes*
CA	*Classified attributes*
C	*Classes*
ECC	*Extensible (non-final) critical classes*
ECM	*Extensible (non-final) classified methods*
CM	*Classified methods*
CSC	*Critical superclasses (top level classes)*
nCSC	*Amount of the classes that may inherit from the critical superclass*
MI	*Classified methods that could be inherited*
AI	*Classified attributes that could be inherited*

Table 6-5 presents the results of the intermediate calculation with which it is possible to calculate results for the multiclass OO metrics.

Table 6-5: Multiclass OO metrics intermediate calculation results

VERSION	CP	CC	cCA	CA	C	ECC	ECM	CM	CSC	nCSC	MI	AI
2.1.0	0	1	16	3	143	1	28	28	1	2	28	3
2.1.1	0	1	16	3	140	1	28	28	1	2	28	3
2.1.2	0	1	16	3	140	1	28	28	1	2	28	3
2.1.3	0	1	16	3	140	1	28	28	1	2	28	3
2.2.0	0	2	19	5	222	2	34	34	2	4	34	5
2.3.0	0	2	19	5	263	2	37	37	2	4	37	5
2.3.1	0	2	19	5	263	2	37	37	2	4	37	5
2.3.2	0	2	19	5	263	2	37	37	2	4	37	5
2.3.2.1	0	2	19	5	264	2	38	38	2	4	38	5

6.3.4 Analysis using the attack surface metric

The attack surface metric uses quite different terms in the calculations than the object-oriented metrics. Table 6-6 explains the terms used in the intermediate calculations. In the following tables, some steps of the attack surface calculation (see Table 5-1) are presented together for better clarity.

Table 6-6: Attack surface calculation terminology

TERM	EXPLANATION
INPUT PARAM	*Method receives or sends data as input parameter*
EXT STORE	*Method receives or sends data to external data source (Java IO commands)*
OTHER SYST	*Method receives or sends data to other systems in the environment*
PUBLIC	*Method is in public interface (other access rights)*
INTERNAL	*Method is in internal interface (access right private)*
CHANNEL COUNT	*The number of channels*

There are a few things to consider when calculating the number of methods, channels and data items, despite much fewer ambiguities than in the previous set of metrics.

Firstly, the process of counting the direct entry points (DEP) included the following:

- DEP, type 1: The number of methods from the OO metrics' results excluding the methods without parameters. Separate the private methods from this amount.
- DEP, type 2: James does not invoke other system's methods, which means there are no values of this type.
- DEP, type 3: Finds methods that use java.io read type methods from Java call graph.

Second, the process of counting the direct exit points (DExP) was the following:

- DExP, type 1: The number of methods from the OO metrics' results excluding the methods with no return value. Separate the private methods from this amount.
- DExP, type 2: James does not invoke other system's methods, which means there are no values of this type.
- DExP, type 3: Finds methods that use java.io write type methods from Java call graph.

Manadhata does not define exactly what methods are considered to be read- or write-type in the Java IO libraries [38]. In this analysis any function type from the IO library that is clearly reading or writing is accounted for, such as 'print' or 'get' methods. Methods that were parts of IO operations were not taken into account, such as 'flush' or 'close' methods.

The number of channels was determined by running the program with the default settings. There were some issues in trying to run some of the James versions. Versions 2.1.0 to 2.2.0 will not run with the modern Java runtime environments and the old runtime environments will not install properly in modern Windows. However, considering the stability of the channel count in other versions, there is no reason to believe the count would be any different in earlier James versions.

The data item count was calculated by inspecting the running program. James does not access any files in the default configuration. With the user file repository configured, the situation might be different, but this would require configuration for the program. Since there were no data items accessed, the data item count was disregarded from the calculations, which is what Manadhata did in his examples [38].

Table 6-7 shows the results of the identification phase (phase 1) and the estimation steps of the phase 2. The columns contain two numbers added together. The first term of the summation is the number of DEPs and the second term of the summation is the number of DExPs. After this phase, only the result of the summation is used, rather than separate values.

Table 6-7: Attack surface calculations phase 1 and partial phase 2 results

VERSION	**INPUT PARAM, PUBLIC**	**INPUT PARAM, INTERNAL**	**EXT STORE, PUBLIC**	**EXT STORE, INTERNAL**	**OTHER SYST, PUBLIC**	**OTHER SYST, INTERNAL**	**CHANNEL COUNT**
2.1.0	527+451	108+52	33+173	19+10	0	0	x
2.1.1	533+447	111+55	36+124	8+57	0	0	x
2.1.2	536+449	110+55	37+128	8+57	0	0	x
2.1.3	539+450	114+55	37+128	8+57	0	0	x
2.2.0	572+598	160+115	44+127	10+50	0	0	x
2.3.0	749+751	197+136	59+135	8+92	0	0	4
2.3.1	749+751	197+136	59+135	8+92	0	0	4
2.3.2	746+756	198+137	59+133	8+92	0	0	4
2.3.2.1	753+754	198+137	59+133	8+92	0	0	4

Table 6-8 presents the rest of phase 2, which includes organizing the potentials and efforts, along with assigning numeric values for them.

Table 6-8: Attack surface calculations phase 2 end results

METHODS

POTENTIAL	**VALUE**	**EFFORT**	**VALUE**
INPUT PARAM	5	PUBLIC	5
OTHER SYST	3	INTERNAL	1
EXT STORE	1		

CHANNELS

POTENTIAL	**VALUE**	**EFFORT**	**VALUE**
TCP	1	USER	5

Manadhata does not define the situation when one should perform calculations for a running program, causing uncertainties for both channel and data item calculations. Does the program have to be fully configured or is a freshly installed program adequate? In this analysis, James was run as freshly installed program without any configuration.

Table 6-9 includes the results of calculation phase 3, excluding the channel calculations from the attack surface results because they have no meaningful effect on the outcome. There are six groups present in the table corresponding to the possible combinations from the Table 6-8 (3 * 2 = 6). The groups are the following:

- Group 1: input parameter, public
- Group 2: input parameter, internal
- Group 3: external store, public
- Group 4: external store, internal
- Group 5: other system public
- Group 6: other system, internal

The attack surface is single number instead of a triple in this calculation. This notation seems to be possible since Manadhata uses it in an example as well [38]. The reason for this notation is clarity, since displaying the two other values that are constants of zero, is pointless.

Table 6-9: Attack surface calculations phase 3 results

VERSION	GROUP 1	GROUP 2	GROUP 3	GROUP 4	GROUP 5	GROUP 6	ATTACK SURFACE
2.1.0	978	800	41,2	29	0	0	1848,2
2.1.1	980	830	32	65	0	0	1907
2.1.2	985	825	33	65	0	0	1908
2.1.3	989	845	33	65	0	0	1932
2.2.0	1170	1375	34,20	60	0	0	2639,2
2.3.0	1500	1665	38,8	100	0	0	3303,8
2.3.1	1500	1665	38,8	100	0	0	3303,8
2.3.2	1502	1675	38,4	100	0	0	3315,4
2.3.2.1	1507	1675	38,40	100	0	0	3320,4

7 RESULTS OF THE CASE STUDY ANALYSIS

This chapter contains the results of the metrics' calculations for Apache James. The final results for the metrics use the intermediate calculation term results from Section 6.3. This chapter also provides answers to RQ3 through analysis of the combined data from Chapter 6 and the appendices. After each metric has its results presented and analyzed separately in its own section, all metrics are analyzed together.

7.1 Results of the single class metrics

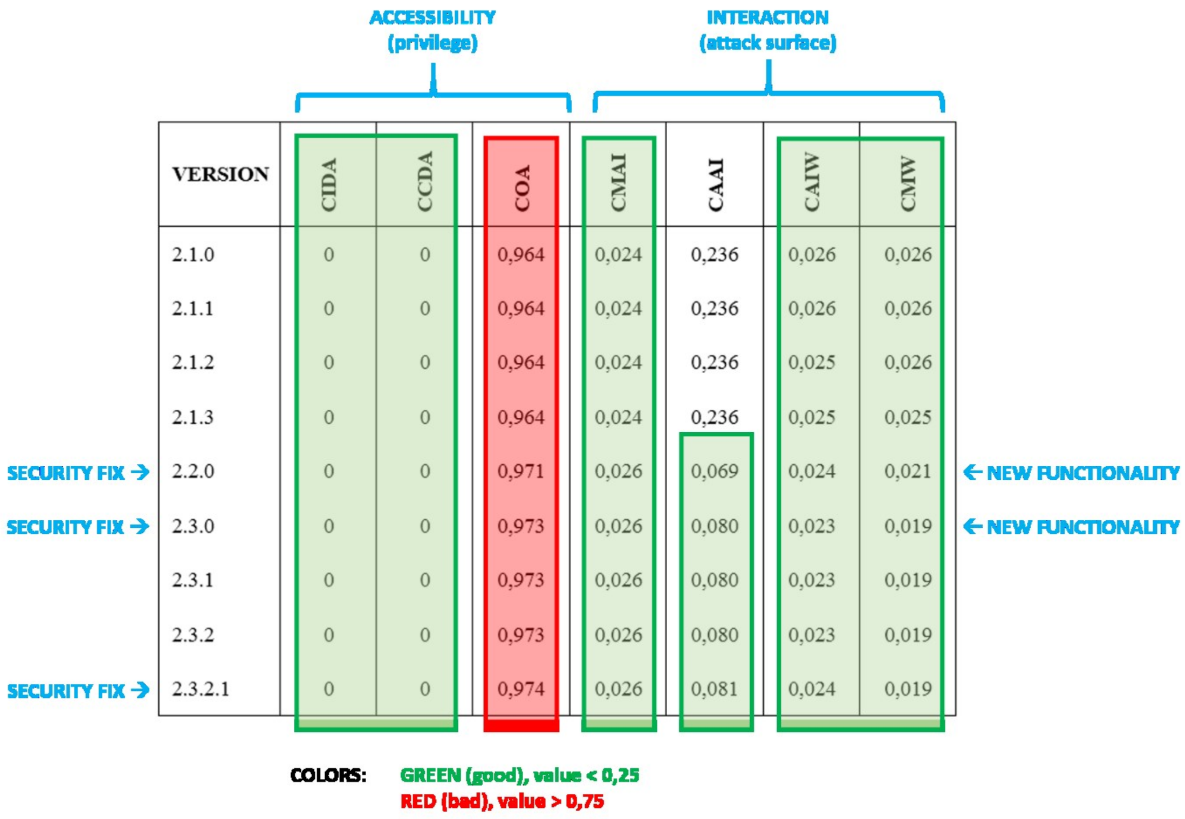
Table 7-1 shows the final calculation results based on the intermediate results from Table 6-3.

Table 7-1: Single class OO metrics' results

VERSION	**CIDA**	**CCDA**	**COA**	**CMAI**	**CAAI**	**CAIW**	**CMW**
2.1.0	0	0	0,964	0,024	0,236	0,026	0,026
2.1.1	0	0	0,964	0,024	0,236	0,026	0,026
2.1.2	0	0	0,964	0,024	0,236	0,025	0,026
2.1.3	0	0	0,964	0,024	0,236	0,025	0,025
2.2.0	0	0	0,971	0,026	0,069	0,024	0,021
2.3.0	0	0	0,973	0,026	0,080	0,023	0,019
2.3.1	0	0	0,973	0,026	0,080	0,023	0,019
2.3.2	0	0	0,973	0,026	0,080	0,023	0,019
2.3.2.1	0	0	0,974	0,026	0,081	0,024	0,019

Since the results are not really self-evident, Figure 7-1 provides some helpful guidance for what the meanings of each value. It is important to recollect that values closer to zero are better in Alshammari's metrics.

A first point to make regarding the results is that many values are consistently the same or strikingly similar. The only metric with substantial change is the CAAI, but the change is the result of adding Accessor Methods with new functionality in version 2.2.0 [Appendix B]. It is quite evident from the formula for CAAI [Table 4-8] that the shift towards more secure state is only spurious. The results seem to indicate that the confidential data is protected and that it interacts very little with anything else. However, the operations interacting with confidential data are open.



VERSION	CIDA	CCDA	COA	CMAI	CAAI	CAIW	CMW
2.1.0	0	0	0,964	0,024	0,236	0,026	0,026
2.1.1	0	0	0,964	0,024	0,236	0,026	0,026
2.1.2	0	0	0,964	0,024	0,236	0,025	0,026
2.1.3	0	0	0,964	0,024	0,236	0,025	0,025
2.2.0	0	0	0,971	0,026	0,069	0,024	0,021
2.3.0	0	0	0,973	0,026	0,080	0,023	0,019
2.3.1	0	0	0,973	0,026	0,080	0,023	0,019
2.3.2	0	0	0,973	0,026	0,080	0,023	0,019
2.3.2.1	0	0	0,974	0,026	0,081	0,024	0,019

ACCESSIBILITY
(privilege)
INTERACTION
(attack surface)
SECURITY FIX →
SECURITY FIX →
SECURITY FIX →
← NEW FUNCTIONALITY
← NEW FUNCTIONALITY
COLORS:
GREEN (good), value < 0,25
RED (bad), value > 0,75

Figure 7-1: Guidance for interpreting the Single class OO metrics' results

The results were affected by any changes close to the data marked confidential, exemplified in the situation with CAAI. Other changes, such as the addition of a large number of methods or attributes, appear to have very small effects on the results. This is evident when looking at differences in results for versions 2.2.0 and 2.3.0, which exhibit major functionality changes.

The COA value strikes out as being bad. Alshammari's description says that COA "helps protect classified internal operations from direct access". COA is the ratio of non-private

CMs to private CMs and since all of the operations on CAs are public, the COA value is high. The COA result raises the question that what kind of class design would have all CMs private (the ideal situation according to COA) since accessing CAs requires non-private CMs.

Alshammari's metrics have a scale ranging from zero (good) to one (bad). However, Alshammari does not provide any instructions on whether the scale should be interpreted as linear or exponential. The connection to the security design principles allows the user to derive general interpretations from the results, but detecting individual security flaws seems to be out of the question.

Alshammari used also Apache James as a target in his work, so for a comparison Table 7-2 shows the results he presented [35] for James v2.1.0 to v2.1.3. Table 7-2 has two additional columns indicating the number of Classified Attributes and Classified Methods used in Alshammari's study.

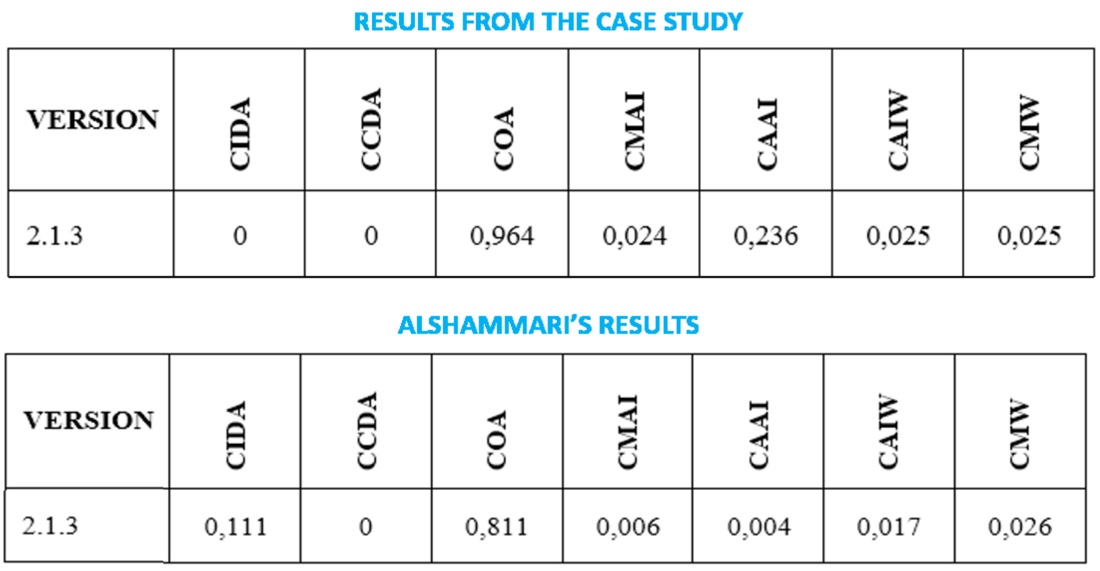
Table 7-2: Alshammari's results [35] for single class OO metrics

VERSION	**CA**	**CM**	**CIDA**	**CCDA**	**COA**	**CMAI**	**CAAI**	**CAIW**	**CMW**
2.1.0	11	57	0,091	0	0,509	0,009	0,010	0,041	0,041
2.1.1	11	57	0,091	0	0,509	0,009	0,009	0,040	0,040
2.1.2	11	57	0,091	0	0,509	0,009	0,010	0,040	0,040
2.1.3	9	37	0,111	0	0,811	0,006	0,004	0,017	0,026

At first glance, Alshammari's values appear different from those in this study. However, in his results there are 11 CAs for version 2.1.0. He mentions an increase in CAs, but increasing the amount of CAs should not be possible without new annotations for confidential data (which he does not do according to his annotation descriptions).

Even with the relaxed interpretation of Classified Methods, the number of CMs in the results of this study are lower than Alshammari's values. The other differences in values likely reflect different definitions of which methods and attributes are counted in each step.

With this in mind, a different picture emerges from the comparison when the differences in CAs and CMs are excluded and the results from James version 2.1.3 are compared side by side. Figure 7-2 does just this, and by comparing the numbers and neglecting scale issues, the results are actually quite close to each other. So based on the results the case study in this thesis has been conducted similarly to the author of the metric. It is troublesome that big visible differences are lacking in the quantitative results despite the CA and CM amounts being different.



RESULTS FROM THE CASE STUDY

VERSION	CIDA	CCDA	COA	CMAI	CAAI	CAIW	CMW
2.1.3	0	0	0,964	0,024	0,236	0,025	0,025

ALSHAMMARI'S RESULTS

VERSION	CIDA	CCDA	COA	CMAI	CAAI	CAIW	CMW
2.1.3	0,111	0	0,811	0,006	0,004	0,017	0,026

Figure 7-2: Comparison of James 2.1.3 case study results with Alshammari's results for Single class OO metrics

Alshammari suggests using a radar chart for graphical representations of the results, but this seems unjustified given the small differences between the versions of James. There seem to be no correlation between the metrics' values and the security flaws [Appendix A] or the major feature changes seen in Figure 7-1.

The single class metrics appear sensitive to the number of CAs and methods accessing them. If they remain constant, this means almost no changes will occur in the metrics' results. The single class metrics are also sensitive to changes in the access right levels in methods, but most of the methods in James are public and have remained thus in new additions. Without change to the access rights, there will be no changes to the results.

7.2 Results of the multiclass metrics

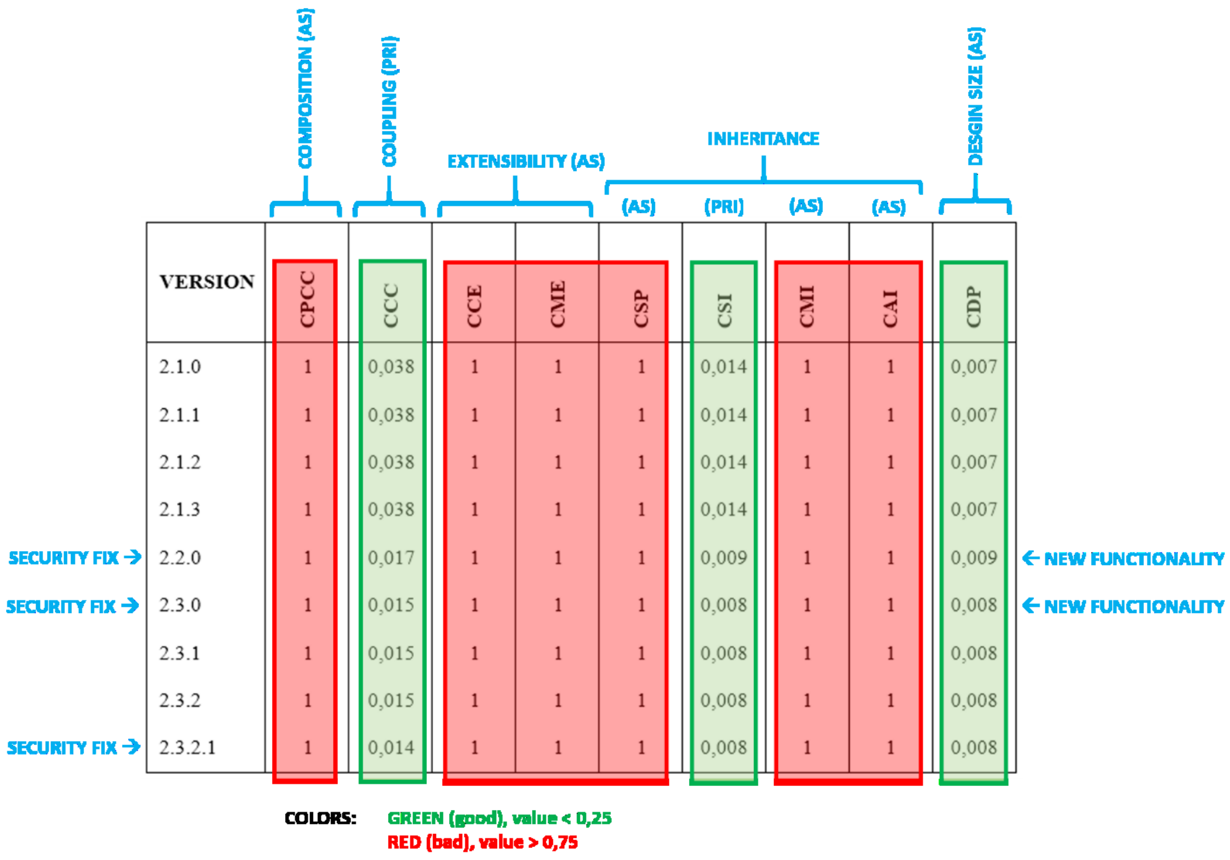
The multiclass metrics are examined in a similar order to the single class metrics from the previous section, with results presented first, which are then compared to Alshammari's results before some concluding remarks. Table 7-3 shows the results for the multiclass metrics.

Table 7-3: Multiclass OO metrics' results

VERSION	**CPCC**	**CCC**	**CCE**	**CME**	**CSP**	**CSI**	**CMI**	**CAI**	**CDP**
2.1.0	1	0,038	1	1	1	0,014	1	1	0,007
2.1.1	1	0,038	1	1	1	0,014	1	1	0,007
2.1.2	1	0,038	1	1	1	0,014	1	1	0,007
2.1.3	1	0,038	1	1	1	0,014	1	1	0,007
2.2.0	1	0,017	1	1	1	0,009	1	1	0,009
2.3.0	1	0,015	1	1	1	0,008	1	1	0,008
2.3.1	1	0,015	1	1	1	0,008	1	1	0,008
2.3.2	1	0,015	1	1	1	0,008	1	1	0,008
2.3.2.1	1	0,014	1	1	1	0,008	1	1	0,008

Although there is very little to interpret, Figure 7-3 provides guidance for interpreting the results, which remain almost entirely unchanged. The bad values are for metrics related to the attack surface security design principle (i.e. CPCC, CCE, CME, CSP, CMI and CAI), meaning that objects (or classes) handing the confidential data are accessible. The CPCC is bad because there are no inner classes in the James class structure. The CCE, CME, CSP, CMI and CAI are bad because all of the CCs can be inherited, they are at the top of inheritance hierarchy, and all of the CMs and CAs can be inherited. The lack of use of object-oriented design principles in James is quite evident from these results but whether it actually signifies any security related conclusions is not so evident. The confidential data seems to be protected since the metrics related to the security design principle of least privilege have good values.

A closer look at the intermediate calculation results from Table 6-5 and the formulas for multiclass metrics (see Table 4-9) reveal that the changes in the numeric values are the result of more classes being added to James. Because no changes are made to class structure around the confidential data, the metrics' results remain unchanged through the versions.


COMPOSITION (AS)
COUPLING (PRI)
EXTENSIBILITY (AS)
INHERITANCE
(AS)
(PRI)
(AS)
(AS)
DESGIN SIZE (AS)
SECURITY FIX →
SECURITY FIX →
SECURITY FIX →
← NEW FUNCTIONALITY
← NEW FUNCTIONALITY
COLORS:
GREEN (good), value < 0,25
RED (bad), value > 0,75

VERSION	CPCC	CCC	CCE	CME	CSP	CSI	CMI	CAI	CDP
2.1.0	1	0,038	1	1	1	0,014	1	1	0,007
2.1.1	1	0,038	1	1	1	0,014	1	1	0,007
2.1.2	1	0,038	1	1	1	0,014	1	1	0,007
2.1.3	1	0,038	1	1	1	0,014	1	1	0,007
2.2.0	1	0,017	1	1	1	0,009	1	1	0,009
2.3.0	1	0,015	1	1	1	0,008	1	1	0,008
2.3.1	1	0,015	1	1	1	0,008	1	1	0,008
2.3.2	1	0,015	1	1	1	0,008	1	1	0,008
2.3.2.1	1	0,014	1	1	1	0,008	1	1	0,008

Figure 7-3: Guidance for interpreting the Multiclass OO metrics' results

For comparison, Table 7-4 presents Alshammari's results for the multiclass metrics for James versions 2.1.0 to 2.1.3. The first column in Table 7-4 shows the number of Critical Classes that Alshammari used, which is higher than that used in this case study. The reason for the higher count in Alshammari's work is the inclusion of more CAs (see discussion in Section 7-1).

Table 7-4: Alshammari's results [35] for multiclass OO metrics

VERSION	CC	CPCC	CCC	CCE	CME	CSP	CSI	CMI	CAI	CDP
2.1.0	4	1	0,007	1	1	1	0,005	1	0	0,020
2.1.1	4	1	0,007	1	1	1	0,005	1	0	0,020
2.1.2	4	1	0,007	1	1	1	0,005	1	0	0,020
2.1.3	5	1	0,008	1	1	0,5	0,003	0,7	0	0,025

The changes in values for CSP and CMI metrics in Table 7-4 between versions 2.1.2 and 2.1.3 are the result of an increase in the amount of CCs. It is interesting to notice that the CAI metric result, which is zero in Alshammari's results and one in the results of this case study. Alshammari's result would indicate no CAs could be inherited, but by reading the code of the classes, there seems to be no way to account for this interpretation.

Presenting a radar chart for the results of the multiclass metrics would be useless due to the unchanging nature of the values. The multiclass metrics seem unable to detect any security flaws or newly added functionalities. In fact, the multiclass metrics seem to have very little to give for software like James, which does not use features of object oriented programming in any large scale.

7.3 Results of the attack surface metric

The attack surface metric's results are a bit different from the OO metrics, since they can be presented in a form suggested by the metric author. Figure 7-4 presents the results in a bar chart form.

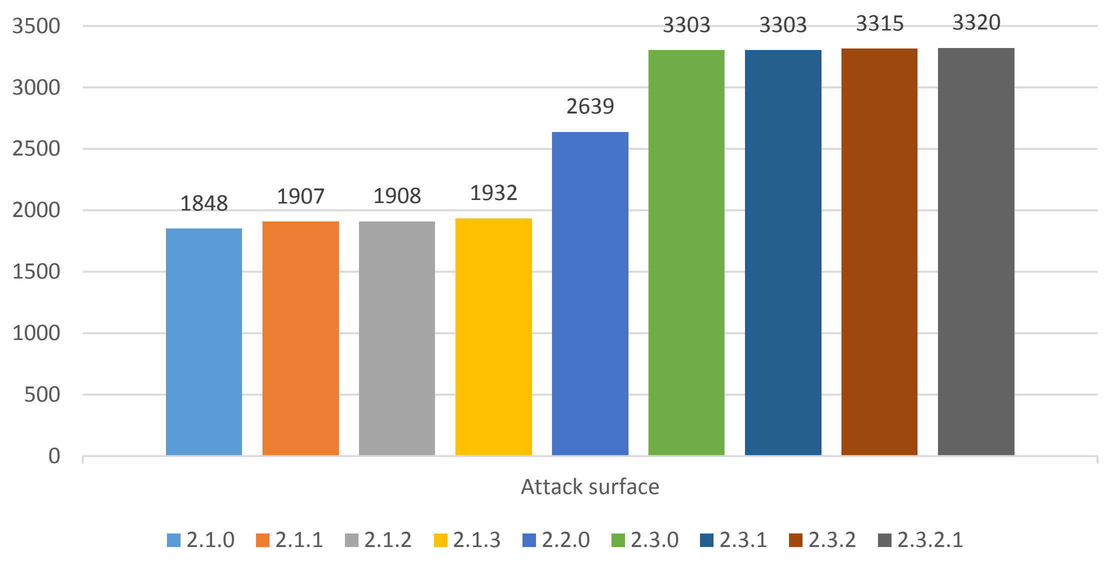


Figure 7-4: Results of the attack surface metric

The bar chart shows the increase in the attack surface clearly. However, the connection to security flaws or new functionality is not apparent. Figure 7-5 depicts how the security flaws and major functionality introductions relate to the attack surface results.

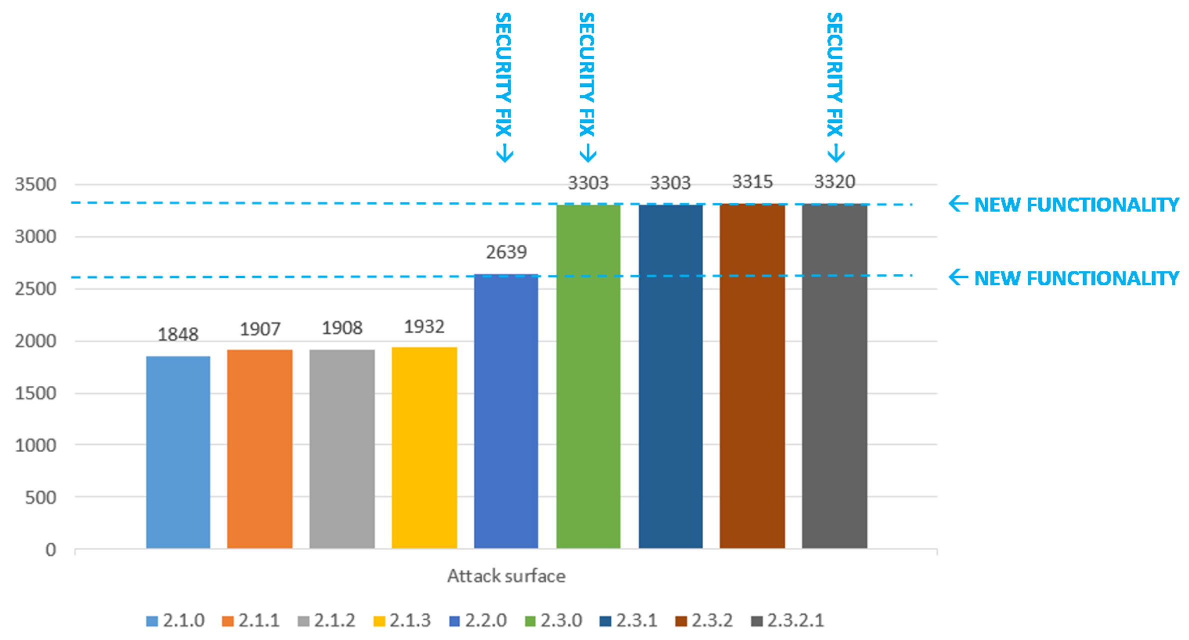


Figure 7-5: The security flaws and new functionality introductions highlighted in the attack surface metric's results

The results of the attack surface metric do not include values for channels or data items (unlike in Table 5-4), since those were absent in the calculations for James. This means that the attack surface value directly measures the number of methods and access rights for these methods.

It is obvious from Figure 7-5 that the metric does not detect the security flaws. The new feature introductions are clearly visible, since they are closely related to the number of methods. The usefulness of the attack surface metric for Java programs like James is questionable, since similar results can be obtained with much simpler/easier metrics as shown in Figure 7-6.

Results of the attack surface metric are compared with Lines of Code (LOC) metric in Figure 7-6. The LOC metric is defined as the amount of lines that have code in James excluding comment and blank lines from the source code files. The comparison of these two metrics shows that in its current form, the attack surface metric is quite redundant.

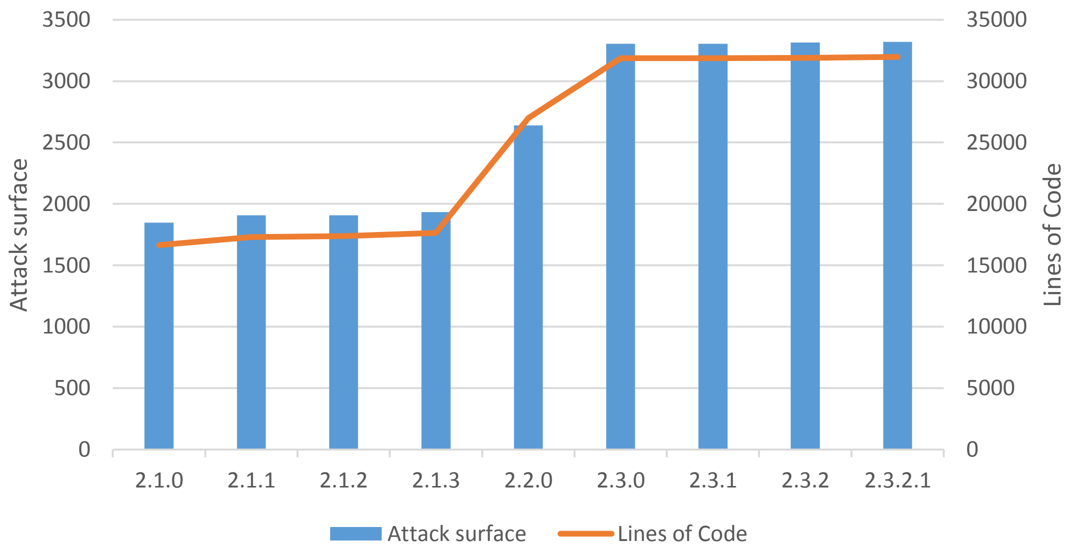


Figure 7-6: Attack surface metric results versus Lines of Code in James

7.4 Examining all results together

Single and Multiclass OO metrics both follow the confidential data closely, which means that a poor initial choice for the annotations has a huge effect on the final results. Finetuning the metrics (i.e. changing the confidential data marking from attributes other attributes) is quite troublesome since calculating results requires somewhat large effort. Perhaps choosing some other data as ‘confidential’ for James would have revealed something different about the software. The metric author did not provide instructions on deciding what should be considered ‘confidential’. Changes in other parts of the program barely showed up in the numeric values provided by the metric.

The attack surface metric did not suffer from annotation issues, but it had multiple issues in the practical application of the metric. Firstly, two out of the three values were unused, while the last remaining value seemed to follow the number of methods closely. The method access right level had an effect on the attack surface calculation result, but in the case of James, most methods are public. Therefore, the metric could simply be replaced by counting lines of code from the program.

All three metrics have their merits and weaknesses, but in general they are not good design time metrics based on multiple factors. The metrics are not applied easily, nor are their results presented in a clear manner, making it difficult to determine strong and weak points in the software design. The metrics do not even work with design time artifacts, but require source code.

Use of source code highlights another problem: it is quite hard to see if the design of software has changed just based on the source code. It is common practice in software engineering to visualize design changes in UML or other graphical charts. Analyzing James source code points to the direction that the design of James remained unchanged in some areas (e.g. user handling) and changed in other areas (e.g. database handling) through the versions analyzed in this thesis, but there are no definite documentation available to verify this observation or the scale of the design changes.

The metrics start with the assumption that they can give quantitative values as results. This idealistic starting point is disturbed by the fact that the metrics require multiple subjective decisions (i.e. qualitative knowledge) before they can be applied in practice. All of the metrics are also relative, meaning that any absolute value is only good for comparing it with another version of the same software.

Finally, the metrics do not really provide instructions on how to interpret the results and the metrics' connection to the security design principles is a bit abstract. Because security is a quality attribute, it would be quite important to understand how to assess this quality attribute in context of numeric results.

8 CONCLUSIONS

This chapter combines and analyzes the results obtained for the three research questions for this thesis. First, Section 8.1 recollects the research outcomes from previous chapters and presents them in order of the research questions. Subsequently, Section 8.2 analyzes and evaluates this work before discussing some future research directions.

8.1 Research outcomes

RQ1: How can we measure the security of software?

The computer security field is full of terminology that have differing definitions across the researchers and many aspects of security. In this thesis a distinction between software- and application-security was made to divide security into separate concepts and focus on the software aspect. Software security was defined as the security for software before it was built (i.e. when it is being developed). Application security was defined as the security for software after it had been built and deployed (i.e. when it is in use).

Another important definition is the concept of 'security' within the realm of software. Security was defined as a quality attribute (i.e. a non-functional requirement) of software, similar to usability or speed. This led us to conclude that a security defect, flaw or bug signifies that the quality attribute of 'security' is not met.

Verification of software security revolves around the concepts of evaluation and assurance. Evaluation was defined as judgements by actors (other than the creators of the software) regarding the security of the software. Assurance was defined as the activity where creators of the software ensure that their software is secure. The activity of assurance was the focus of this thesis due to the scope targeting those software engineers who are creating software.

When attempting to measure 'assurance' activity, this work utilized the measurement theory and tried to find suitable metrics for creating a theoretical and practical framework for the assurance activities.

RQ2: What metrics are available for measuring software security?

A total of 34 different software security metrics were found, of which, five were evaluation metrics and 29 of them were assurance metrics. The quality of the metrics varied greatly with many of the metrics being concepts without a theoretical basis or practical instructions for applications. Some of the metrics were not concepts, but rather, discussion papers which ended up suggesting useless procedures for measuring security. However, by categorizing the metrics and inspecting their properties, three design level assurance metrics were found and deemed suitable for further inspection.

The three metrics found were within PhD dissertations and fulfilled an inspection criteria based on measurement theory. They were therefore studied in additional detail so that they could be applied to a case study. Two of the metrics targeted object-oriented programming concepts while the other metric defined software as a mathematical model.

Closer study of the metrics, both in theory and practice, revealed some issues with the metrics. Theoretically, no metric was actually able to use design time artifacts in its measurement process. In practice, this means that rather than using UML charts, one must use source code. In the case of object-oriented metrics, the source code must have specific annotations as well.

Another theoretical problem for all of the metrics is indirect referencing, which is a well-known problem in object-oriented programming that is discussed widely across literature. For example, when discussing aliasing [80] or the specific phenomenon called representational exposure [81]. A simple description of the problem for object-oriented programming is that apart from directly referencing an object, there are ways of indirectly referring to them, thereby bypassing the restrictions set by the programmer.

One of the practical issues that the metrics face is a lack of clarity regarding what calculation values cause changes in the results of the metrics. This means that seemingly unrelated changes in metrics' calculation values might affect a metric result value attempting to describe a different phenomenon. This is especially apparent in object-oriented design metrics, but also appears in the attack surface metric, which seems closely correlated with the number of methods.

The other major practical issue with the metrics is the need for qualitative estimations and subjective decisions in their practical applications. The metric authors downplay the effect of these decisions, but in reality they have a major impact on the metrics' numeric results and how the software is processed using the metrics' instructions.

Based on the results of this thesis, the general state of assurance metrics is not good. This thesis did not inspect the evaluation metrics, but evaluation metrics seemed to be much more mature and are being used in the industry all the time. However, they are designed for governmental or military use, making them troublesome and expensive to use in normal software engineering activities.

RQ3: What do the software security metrics reveal about the security of software?

This thesis conducted a case study research to test specifically chosen assurance metrics. Based on the results, the metrics have large issues in providing any information about the security of software.

The validity of the metric results is questionable because the metrics are relative. In this context, 'relativity' means that the metrics provide meaningful results for only software designs of similar type. It is unclear what characterizes 'similar' types of software was and is thus not clear what would cause the status of a software to be classified as a 'similar' or 'different' type. The attack surface metric also seemed to closely follow the source code line count, which brings into question the usefulness of that particular metric.

All of the metrics connected their theory into the security design principles, but this connection was quite abstract when trying to interpret the numeric results provided by the metrics. None of the metrics could detect security flaws found in the target software, and only the attack surface metric could detect new functionality added to the target software.

8.2 Discussion

What does all of the work done in this thesis mean?

As stated in [18] when it is unclear how to measure an attribute of software (e.g. security), merely attempting to do so will increase understanding of the phenomenon in spite of claims that say non-functional requirements of a software are not quantifiable [18].

The current generation of security metrics seems unsuitable for the development of secure software. However, despite their flaws, the three metrics examined here provide a foundation for future research in software security. Without this work, understanding benefits and drawbacks of the metrics would be much harder. That said, in order to develop secure software one has to use highly subjective and qualitative methods, which include expert reviews such as STRIDE threat modeling [82].

Did the work succeed in its goals?

This work succeeded in the primary goals of discovering the current status of software security metrics and establishing the level of current understanding of secure software, even though the final results were not evaluations of the software security from a design perspective. However, the case study could have used another product that was more security-oriented to provide additional insight into the metrics, but the search for a suitable target was dropped due to time constrains.

One might also wonder if Apache James was a suitable target for the case study for the reasons that it does not have many public security issues, the changelog isn't clearly connected to the source code, and developer documentation is scarce. Regardless of these deficiencies, the case study found multiple areas of interest in the metrics and provided meaningful interpretations from the results.

Limitations of this work

This thesis had only one target software in the case study. Having only a single target limits the results of the case study to be insights. For more conclusive evidence more targets and different targets should be examined.

The metrics had no instructions for practical applications that lead to numerous subjective decisions in how to apply the metrics. These choices made in the details had a considerable role in the outcome of this work. Naturally anyone else attempting to use the metrics will face the same obstacles and has to deal with them.

Possible directions for future work

Contemporary software security measurement seems to be in the hands of expert reviews and code level tools. Further development of these expert review tools could provide fruitful insights and allow for better measurement or assessment of the security of the design. This would avoid some problems arising from quantifying quality attributes such as security.

In order to further develop existing software design security metrics, a stronger connections must be made between the metric's theory and fundamental concepts of software design. The connection between theory and practice also needs to be revisited, as the metrics seem unable to move from theoretical models into real world programming languages.

APPENDIX A: KNOWN VULNERABILITIES OF APACHE JAMES

Table A-1: Details of CVE-2004-2650 (adapted from [83], [84] and [85])

CVE-2004-2650	
Version affected	James < 2.2.0
Vulnerability type	Denial of Service
Description	The Spooler component fails to check for error conditions in a mail retrieval situation, causing a memory leak.
Exploitation	An attacker could create multiple error conditions and eventually consume the system's resources.
End result	Successful exploitation will ultimately crash the application denying service to legitimate users.

Code example	`if (lock(s)) {`
	`    MailImpl mail = null;`
	`    try`
	`        { mail = retrieve(s); }`
	`    catch (javax.mail.MessagingException e)`
	`        { ... }`
	`    if (mail == null)`
	`        { continue; }`
	`}`
	
	`If retrieve returns null or throws an exception, the lock is kept, and we leak memory.`

Table A-2: Details of CVE-2006-2806 (adapted from [86], [87] and [88])

CVE-2006-2806	
Version affected	James < 2.3.0
Vulnerability type	Denial of Service
Description	The SMTP component fails to process malformed SMTP commands in a sensible manner, causing the program to consume a great amount of CPU cycles.
Exploitation	An attacker could use the standard network tools to connect to the SMTP port and enter malformed commands, causing a great CPU load.
End result	A successful exploitation will ultimately cause the application to be slow or unresponsive for legitimate users.
Code example	`$socket = IO::Socket::INET->new(Proto=>"tcp", PeerAddr=>$host, PeerPort=>"25", Reuse=>1)` `while ( $i++ ) {` `print $socket "MAIL FROM:" . "fvclz" x 1000000 . "\r\n" and` `print " -- sucking CPU resources at $host\n";` `}`

```
$socket = IO::Socket::INET->new(Proto=>"tcp",
PeerAddr=>$host, PeerPort=>"25", Reuse=>1)

while ( $i++ ) {
    print $socket "MAIL FROM:" . "fvclz" x 1000000 .
"\r\n" and
    print " -- sucking CPU resources at $host\n";
}
```

Table A-3: Details of CVE-2015-7611 (adapted from [89], [90] and [91])

CVE-2015-7611	
Version affected	James < 2.3.2
Vulnerability type	Remote command execution
Description	The user addition function fails to correctly process illegal usernames thereby opening exploitation avenues in file based user repositories.
Exploitation	An attacker requires legitimate access to the remote administration tool. Within the tool an attacker can create a user with the username of a file and afterwards send an email to this particular "user" describing the contents of the file.
End result	Successful exploitation will ultimately allow the attackers to execute arbitrary system commands within the context of the application.

Code example

```
print "[+]Connecting to Remote Administratio Tool"
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

print "[+]Creating user"
s.send("adduser
../../../../../../../../etc/bash_completion.d ex-
ploit\n")

print "[+]Connecting to James SMTP server"
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.send("ehlo team@team.pl\r\n")

print "[+]Sending payload..."
s.send("mail from: <'@team.pl>\r\n")
s.send("rcpt to:
<../../../../../../../../etc/bash_comple-
tion.d>\r\n")
s.send("From: team@team.pl\r\n")
s.send("\r\n")
s.send("'\n")
s.send(payload + "\n")
s.send("\r\n.\r\n")
s.send("quit\r\n")

print "[+]Done! Payload will be executed once some-
body logs in."
```

APPENDIX B: MAJOR CHANGES BETWEEN APACHE JAMES VERSIONS

Version 2.1.0 [92] 29 December 2002

- major SMTP feature updates
- fixes to POP3 message engine
- added NNTP support

Version 2.1.1 [92] 11 February 2003

- fixes synchronization issues

Version 2.1.2 [92] 21 February 2003

- fixes fatal connection errors and the bounce mechanism

Version 2.1.3 [92] 12 May 2003

- spooler fixes
- nntp fixes

Version 2.2.0 [92] 15 June 2004

- fixes CVE-2004-2650
- mailbox system created
- added support for mail redirecting
- added support for remote gateway email servers
- new Mailets
- some external library updates

Version 2.3.0 [93] 23 October 2006

- fixes CVE-2006-2806
- SMTP server functionality upgrades
- changes in the database engine
- new filters and Mailets
- many external library updates

Version 2.3.1 [93] 29 April 2007

- license information update
- bugfixes to external components, notably Sendmail plugin

Version 2.3.2 [94] 10 August 2009

- a few minor bug fixes and external library updates

Version 2.3.2.1 [95] 30 September 2015

- fixes CVE-2015-7611

REFERENCES

[1] M. Bishop, "What is computer security?," *IEEE Security & Privacy,* pp. 67-69, January 2003.

[2] G. McGraw, "Building Security In - Software Security," *IEEE Security & Privacy,* pp. 80-83, April 2004.

[3] F. Nunes, A. Belchior and A. Albuquerque, "Security Engineering Approach to Support Software Security," in *6th World Congress on Services*, Miami, 2010.

[4] B. Alshammari, C. Fidge and D. Corney, "Security Metrics for Object-Oriented Class Designs," in *Proceeding of QSIC 2009 : Ninth International Conference on Quality Software*, Jeju, 2009.

[5] J. Zalewski, S. Drager and A. Kornecki, "Can We Measure Security and How?," in *Seventh Annual Workshop on Cyber Security and Information Intelligence Research*, Oak Ridge, 2011.

[6] A. Jaquith, Security metrics, Upper Saddle River: Pearson Education Inc., 2007.

[7] B. Bruegge and A. Dutoit, Object-Oriented Software Engineering, Upper Saddle River: Prentice Hall, 2010.

[8] M. Kainerstorfer, J. Sametinger and A. Wiesauer, "Software Security for Small Development Teams - A Case Study," in *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, Ho Chi Minh City, 2011.

[9] Microsoft, Security Development Lifecycle Process Guidance Version 5.2, 2012.

[10] US National Institute of Standards and Technology, "The Economic Impacts of Inadequate Infrastructure for Software Testing (NIST Planning Report 02-3)," 2002.

[11] T. Klein, "The myth of the cost of defect," 21 May 2015. [Online]. Available: http://thklein.com/en_US/cost-of-defect/. [Accessed 22 October 2015].

[12] C. Jones, "A short history of the cost per defect metric," 5 May 2009. [Online]. Available: http://blog.paluno.uni-due.de/semat.org/wp-content/uploads/2012/03/a_short_history_of_the_cost_per_defect_metric.doc. [Accessed 22 October 2015].

[13] C. Jones and O. Bonsignour, The Economics of Software Quality, Boston: Pearson Education, 2012.

[14] R. Anderson, Security Engineering, Second edition, Wiley, 2008.

[15] ISO/IEC, "Systems and software engineering - System life cycle processes (ISO/IEC/IEEE 15288:2008)," 2008.

[16] Common Criteria Recognition Arrangement, "Common Criteria for Information Technology Security Evaluation, versio 3.1 release 4," 2012.

[17] J. Zalewski, S. Drager and A. Kornecki, "Measuring Security: A Challenge for the Generation," in *Position Papers of the 2014 Federated Conference on Computer Science and Information Systems*, Warsaw, 2014.

[18] N. E. Fenton and S. L. Pfleeger, Software Metrics - A Rigorous and Practical Approach, Boston: PWS Publishing Company, 1997.

[19] T. Heyman, R. Scandariato, C. Huygens and W. Joosen, "Using security patterns to combine security metrics," in *Proceedings of the Third International Conference on Availability, Security and Reliability*, Barcelona, 2008.

[20] US National Institute of Standards and Technology, "Directions in Security Metrics Research (NIST IR 7564)," 2009.

[21] D. Mellado, E. Fernández-Medina and M. Piattini, "A Comparison of Software Design Security Metrics," in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, Copenhagen, 2010.

[22] US Department of Defence, "Trusted Computer System Evaluation Criteria, DoD 5200.28-STD," 1985.

[23] Commission of the European Communities Directorate XIII/F SOG-IS, "Information Technology Security Evaluation Criteria (ITSEC)," 1991.

[24] S. Lipner, "The Birth and Death of the Orange Book," *IEEE Annals of the History of Computing,* pp. 19-31, April-June 2015.

[25] US National Institute of Standards and Technology, "Performance Measurement Guide for Information Security (NIST 800-55)," 2008.

[26] C. Du, X. Li, H. Shi, J. Hu, R. Feng and Z. Feng, "Architecture Security Evaluation Method based on Security of the Components," in *Software Engineering Conference (APSEC)*, Bangkok, 2013.

[27] ISO/IEC, "Information security management - Measurement (ISO/IEC 27004:2009)," 2009.

[28] Swedish Defence Research Agency, "Design and Use of Information Security Metrics," Linköping, 2011.

[29] B. Rodes, J. Knight and K. Wasson, "A Security Metric Based on Security Arguments," in *WETSoM 2014: Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*, Hyderabad, 2014.

[30] ISO/IEC, "Systems Security Engineering - Capability Maturity Model (ISO/IEC 21827:2008)," 2014.

[31] K. Ferraiolo, "National Information Systems Security Conference," 2000. [Online]. Available: http://csrc.nist.gov/nissc/2000/proceedings/papers/916slide.pdf. [Accessed 8 October 2015].

[32] S. Khan and R. Khan, "Security assessment framework: a complexity perspective," *Computer Fraud & Security,* pp. 13-17, July 2014.

[33] S. Chandra, R. A. Khan and A. Agrawal, "Security Estimation Framework: Design Phase Perspective," in *ITNG '09: Sixth International Conference on Information Technology: New Generations*, Las Vegas, 2009.

[34] K. Sultan, A. En-Nouaary and A. Hamou-Lhadj, "Catalog of Metrics for Assessing Security Risks of Software throughout the Software Development Life Cycle," in *ISA 2008: International Conference on Information Security and Assurance*, Busan, 2008.

[35] B. Alshammari, Quality Metrics for Assessing Security-Critical Computer Programs, Queensland University of Technology, 2011.

[36] P. Manadhata, K. Tan, R. Maxion and J. Wing, "An Approach to Measuring a System's Attack Surface," Carnegie Mellon University, Pittsburgh, 2007.

[37] S. Jajodia, A. Ghosh, V. Swarup, C. Wang and X. S. Wang, Moving Target Defense, Springer, 2011.

[38] P. Manadhata, An Attack Surface Metric, Carnegie Mellon University, 2008.

[39] MITRE Corporation, "Common Weakness Enumeration FAQ," 25 July 2014. [Online]. Available: https://cwe.mitre.org/about/faq.html. [Accessed 9 October 2015].

[40] FIRST.Org Inc., "Common Vulnerability Scoring System, version 3.0," 2015.

[41] US National Institute of Standards and Technology, "The Common Misuse Scoring System (CMSS): Metrics for Software Feature Misuse Vulnerabilities (NIST IR 7864)," 2012.

[42] J. A. Wang, H. Wang, M. Guo and M. Xia, "Security Metrics for Software Systems," in *Proceedings of the 47th Annual Southeast Regional Conference*, Clemson, 2009.

[43] I. Chowdhury, B. Chan and M. Zulkernine, "Security Metrics for Source Code Structures," in *Proceedings of the fourth international workshop on Software engineering for secure systems*, Leipzig, 2008.

[44] T. Heyman, A Formal Analysis Technique for Secure Software Architectures, KU Leuven, 2013.

[45] B. Berger, K. Sohr and R. Koschke, "Extracting and Analyzing the Implemented Security Architecture of Business Applications," in *CSMR: 17th European Conference on Software Maintenance and Reengineering*, Genova, 2013.

[46] K. Sohr and B. Berger, "Idea: Towards Architecture-Centric Security Analysis of Software," in *Second International Symposium ESSoS*, Pisa, 2010.

[47] P. Carvalho, "Mapping the Software Error and Effects Analysis to ISO 26262 requirements for software architecture analysis," in *IEEE International Symposium on Software Reliability Engineering Workshops*, Naples, 2014.

[48] J. Fragola and J. Spahn, "The Software Error Effects Analysis - A Qualitative Design Tool," in *IEEE Symposium on Computer Software Reliability*, New York, 1973.

[49] B. Alshammari, C. Fidge and D. Corney, "A Hierarchical Security Assessment Model for Object-Oriented Programs," in *Proceedings of the 11th International Conference on Quality Software (QSIC 2011)*, Madrid, 2011.

[50] B. Alshammari, C. Fidge and D. Corney, "Security metrics for object-oriented designs," in *Proceedings of the 21st Australian Software Engineering Conference (ASWEC 2010)*, Auckland, 2010.

[51] Y. Deng, J. Wang and J. Tsai, "Formal Analysis of Software Security System Architectures," in *5th International Symposium on Autonomous Decentralized Systems*, Dallas, 2001.

[52] M. Almorsy, J. Grundy and A. Ibrahim, "Automated Software Architecture Security Risk Analysis using Formalized Signatures," in *ICSE '13: Proceedings of the 2013 International Conference on Software Engineering*, San Francisco, 2013.

[53] S. Halkidis, N. Tsantalis, A. Chatzigeorgiou and G. Stephanides, "Architectural Risk Analysis of Software Systems Based on Security Patterns," *IEEE Transactions on Dependable and Secure Computing,* pp. 129 - 142, July-September 2008.

[54] Y. Liu, I. Traore and A. Hoole, "A Service-Oriented Framework for Quantitative Security Analysis of Software Architectures," in *APSCC '08: Asia-Pacific Services Computing Conference*, Yilan, 2008.

[55] V. Sharma and K. Trivedi, "Architecture Based Analysis of Performance, Reliability and Security of Software Systems," in *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, Palma de Mallorca, 2005.

[56] C. Gegick, Predicting Attack-prone Components with Source Code, North Carolina State University, 2009.

[57] S.-T. Lai, "An Analyzer-based Software Security Measurement Model for Enhancing Software System Security," in *WCSE: Second World Congress on Software Engineering*, Wuhan, 2010.

[58] Y. Shin, A. Meneely, L. Williams and J. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *IEEE Transactions on Software Engineering,* pp. 772 - 787, September 2010.

[59] J. Jürjens, Secure Systems Development with UML, Springer, 2005.

[60] J. Barnes, SPARK: The Proven Approach to High Integrity Software, Altran Praxis, 2012.

[61] AdaCore, "Applying SPARK in Practice," [Online]. Available: http://docs.adacore.com/spark2014-docs/html/ug/usage_scenarios.html. [Accessed 15 November 2015].

[62] AdaCore, "Introduction to SPARK," [Online]. Available: http://docs.adacore.com/spark2014-docs/html/ug/introduction.html. [Accessed 15 November 2015].

[63] Research Group: Software Engineering for Critical Systems, "UML Analysis Tools," Technical University Dortmund, [Online]. Available: https://www-secse.cs.tu-dortmund.de/jj/umlsectool/index.html. [Accessed 15 November 2015].

[64] Research Group: Software Engineering for Critical Systems, "CARiSMA," Technical University Dortmund, [Online]. Available: https://www-secse.cs.tu-dortmund.de/carisma/index.shtml. [Accessed 15 November 2015].

[65] B. Alshammari, C. Fidge and D. Corney, "An Automated Tool for Assessing Security-Critical Designs and Programs," in *Proceedings of WIAR '2012; National Workshop on Information Assurance Research*, Riyadh, 2012.

[66] G. Gousios, "Java Call Graph Utilities," 16 April 2013. [Online]. Available: https://github.com/gousiosg/java-callgraph. [Accessed 15 November 2015].

[67] M. Russinovich, "Process Monitor," Microsoft, 26 May 2015. [Online]. Available: https://technet.microsoft.com/en-us/sysinternals/processmonitor.aspx. [Accessed 15 November 2015].

[68] M. Russinovich, "Process Explorer," Microsoft, 10 March 2015. [Online]. Available: https://technet.microsoft.com/en-us/sysinternals/processexplorer. [Accessed 15 November 2015].

[69] P. Runeson, M. Höst, A. Rainer and B. Regnell, Case Study Research in Software Engineering, New Jersey: John Wiley & Sons, 2012.

[70] The Apache Software Foundation, "James Project," [Online]. Available: http://james.apache.org/. [Accessed 16 November 2015].

[71] The Apache Software Foundation, "Board of Directors Meeting Minutes, 22nd of January 2003," 22 January 2003. [Online]. Available:

http://www.apache.org/foundation/records/minutes/2003/board_minutes_2003_01_22.txt. [Accessed 11 December 2015].

[72] C. Duguay, "Working with James, Part 1: An introduction to Apache's James enterprise e-mail server," Capital Stream Inc., 10 June 2003. [Online]. Available: http://www.ibm.com/developerworks/library/j-james1/. [Accessed 16 November 2015].

[73] The Apache Software Foundation, "Board of Directors Meeting Minutes extracts concerning Apache James," [Online]. Available: https://whimsy.apache.org/board/minutes/JAMES.html. [Accessed 11 December 2015].

[74] M. Nash, Java Frameworks and Components, Cambridge: Cambridge University Press, 2003.

[75] The Apache Software Foundation, "The Apache Avalon Project Closure Notice," 2004. [Online]. Available: http://avalon.apache.org/closed.html. [Accessed 11 December 2015].

[76] D. Angus, "Apache James - The Complete Email Application Platform," 19 July 2010. [Online]. Available: http://www.slideshare.net/Tess98/apache-james. [Accessed 11 December 2015].

[77] The Apache Software Foundation, "Apache James Download Repository," [Online]. Available: http://archive.apache.org/dist/james/server/. [Accessed 16 November 2015].

[78] The WinMerge project, "WinMerge," [Online]. Available: http://winmerge.org/. [Accessed 14 December 2015].

[79] S. Küng, "grepWin," [Online]. Available: http://stefanstools.sourceforge.net/grepWin.html. [Accessed 14 December 2015].

[80] J. Hogg, D. Lea, A. Wills, D. deChampeaux and R. Holt, "The Geneva convention on the treatment of object aliasing," *ACM SIGPLAN OOPS Messenger,* pp. 11-16, 1 April 1992.

[81] D. G. Clarke, J. Noble and J. Potter, "Overcoming Representation Exposure," in *Proceedings of the Workshop on Object-Oriented Technology*, 1999.

[82] S. Hernan, S. Lambert, T. Ostwald and A. Shostack, "Uncover Security Design Flaws Using The STRIDE Approach," *MSDN Magazine,* November 2006.

[83] CVE Details, "CVE-2004-2650," [Online]. Available: http://www.cvedetails.com/cve/CVE-2004-2650/. [Accessed 13 December 2015].

[84] The Apache Software Foundation, "Apache James bug JAMES-268," 22 April 2004. [Online]. Available: https://issues.apache.org/jira/browse/JAMES-268. [Accessed 13 December 2015].

[85] Security Focus, "Apache James Spooler Memory Leak Denial of Service Vulnerability," [Online]. Available: http://www.securityfocus.com/bid/15765/. [Accessed 13 December 2015].

[86] CVE Details, "CVE-2006-2806," [Online]. Available: http://www.cvedetails.com/cve/CVE-2006-2806/. [Accessed 13 December 2015].

[87] The Apache Software Foundation, "Apache James bug JAMES-535," 15 June 2006. [Online]. Available: https://issues.apache.org/jira/browse/JAMES-535. [Accessed 13 December 2015].

[88] Security Focus, "Apache James SMTP Denial Of Service Vulnerability," [Online]. Available: http://www.securityfocus.com/bid/18138/. [Accessed 13 December 2015].

[89] Openwall, "Email exchange on oss-security mailing list," 1 October 2015. [Online]. Available: http://www.openwall.com/lists/oss-security/2015/10/01/2. [Accessed 13 December 2015].

[90] Security Focus, "Apache James Server Unspecified Command Execution Vulnerability," [Online]. Available: http://www.securityfocus.com/bid/76933/. [Accessed 13 December 2015].

[91] Exploit Database, "Apache James Server 2.3.2 Authenticated User Remote Command Execution," [Online]. Available: https://www.exploit-db.com/exploits/35513/. [Accessed 13 December 2015].

[92] The Apache Software Foundation, "Apache James Changelogs, 0.9.5 - 2.2.0," 19 November 2009. [Online]. Available: https://james.apache.org/server/2.2.0/changelog.html. [Accessed 13 December 2015].

[93] The Apache Software Foundation, "Apache James Changelogs, after 2.2.0 - 2.3.1," 30 April 2007. [Online]. Available: https://james.apache.org/server/2.3.1/changelog.html. [Accessed 13 December 2015].

[94] The Apache Software Foundation, "Apache James Changelog, 2.3.2," 9 February 2009. [Online]. Available: http://james.apache.org/server/2.3.2/release-notes.html. [Accessed 13 December 2015].

[95] The Apache Software Foundation, "Apache James Blog on 2.3.2.1," 30 September 2015. [Online]. Available: https://blogs.apache.org/james/entry/apache_james_server_2_3. [Accessed 13 December 2015].