

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE
DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

Static Analysis for Facilitating Secure and Reliable Software

Miltiadis Siavvas

Supervised by Prof. Erol Gelenbe

A thesis submitted in fulfillment of requirements for the degree of
Doctor of Philosophy of Imperial College London and the
Diploma of Imperial College London, April 2019

Abstract

Software security and reliability are aspects of major concern for software development enterprises that wish to deliver dependable software to their customers. Several static analysis-based approaches for facilitating the development of secure and reliable software have been proposed over the years. The purpose of the present thesis is to investigate these approaches and to extend their state of the art by addressing existing open issues that have not been sufficiently addressed yet.

To this end, an empirical study was initially conducted with the purpose to investigate the ability of software metrics (e.g., complexity metrics) to discriminate between different types of vulnerabilities, and to examine whether potential interdependencies exist between different vulnerability types. The results of the analysis revealed that software metrics can be used only as weak indicators of specific security issues, while important interdependencies may exist between different types of vulnerabilities. The study also verified the capacity of software metrics (including previously uninvestigated metrics) to indicate the existence of vulnerabilities in general.

Subsequently, a hierarchical security assessment model able to quantify the internal security level of software products, based on static analysis alerts and software metrics is proposed. The model is practical, since it is fully-automated and operationalized in the form of individual tools, while it is also sufficiently reliable since it was built based on data and well-accepted sources of information. An extensive evaluation of the model on a large volume of empirical data revealed that it is able to reliably assess software security both at product- and at class-level of granularity, with sufficient discretion power, while it may be also used for vulnerability prediction. The experimental results also provide further support regarding the ability of static analysis alerts and software metrics to indicate the existence of software vulnerabilities.

Finally, a mathematical model for calculating the optimum checkpoint interval, i.e., the checkpoint interval that minimizes the execution time of software programs that adopt the application-level checkpoint and restart (ALCR) mechanism was proposed. The optimum checkpoint interval was found to depend on the failure rate of the application, the execution cost for establishing a checkpoint, and the execution cost for restarting a program after failure. Emphasis was given on programs with loops, while the results were illustrated through several numerical examples.

*Dedicated to my beloved parents, Georgios Siavvas and Eleni Chatzivgeri,
and brother, Stefanos Siavvas for their endless love, support, and encouragement*

Acknowledgements¹

First and foremost, I would like to express my deep gratitude and sincere appreciation to my principal supervisor, Professor Erol Gelenbe, for his priceless support and care throughout my studies. It was an honor and a real privilege for me to work under the supervision of a top scientist with renowned contributions in the broader scientific community. I strongly believe that without his invaluable academic guidance and support the present work would not have been possible. Under his supervision, I gained valuable knowledge and developed important skills that significantly helped me in becoming a better researcher. In addition, his advice and guidance have taught me well beyond the academic world, and significantly contributed to shaping my future life.

I would also like to thank my collaborative supervisor at the Information Technologies Institute (ITI), Dr. Dimitrios Tzovaras, for his supervision and support throughout the whole period of my research. His constant support and constructive comments were valuable for the present thesis. Furthermore, I wish to thank Dr. Dionysios Kehagias (ITI) for his valuable assistance with his knowledge and experience.

Last but not least, I would like to thank my parents, Georgios Siavvas and Eleni Chatzivgeri, and also my brother, Stefanos Siavvas for their continuous and unparalleled love, help, and support throughout all the years of my studies. This journey would not have been possible without them, and therefore I dedicate this success to them.

¹A major part of this thesis was supported by the European Union's Horizon 2020 Research and Innovation Programme through SDK4ED project under Grant Agreement No. 780572, as well as through CloudLightning project under Grant Agreement No. 643946. The opinions expressed in this thesis are those of the author and do not necessarily reflect the views of the European Commission.

Abbreviations

AHP	Analytic Hierarchy Process
ALCR	Application-level Checkpoint and Restart
ANOVA	Analysis of Variance
ASA	Automatic Static Analysis
ASM	Attack Surface Metric
ASSM	Analyzer-based Software Security Measurement
Ca	Afferent Coupling
CAM	Cohesion Among Methods
CASE	Computer-aided Software Engineering
CBM	Coupling Between Methods
CBO	Coupling Between Objects
CCC	Complexity Coupling and Cohesion
CCP	Coupling Code Propagation
Ce	Efferent Coupling
CER	Critical Element Ratio
CK	Chidamber and Kemerer
CR	Checkpoint and Restart
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DAM	Data Access Metric
DP-E	Damage Potential-Effort
FPR	False Positive Rate
GUI	Graphic User Interface
HPC	High Performance Computing
IC	Inheritance Coupling
IEC	International Engineering Consortium
ISO	International Standardization Organization

LCOM	Lack of Cohesion in Methods
LOC	Lines of Code
MFA	Measure of Functional Abstraction
ML	Machine Learning
MPI	Message Passing Interface
NOC	Number of Children
NPM	Number of Public Methods
NVP	N-Version Programming
OCI	Optimum Checkpoint Interval
OCR	Optimum Checkpoint Recommendation
OWASP	Open Web Application Security Project
PWC	Pairwise Comparison
QATCH	Quality Assessment Tool Chain
QoS	Quality of Service
QSA	Quantitative Security Assessment
RASQ	Relative Attack Surface Quotient
RB	Recovery Block
RQ	Research Question
SaaS	Software-as-a-Service
SAVD	Static Analysis Vulnerability Density
SAVI	Static Analysis Vulnerability Indicator
SCA	Static Code Analysis
SIG	Software Improvement Group
SLOC	Source Lines of Code
SR	Stall Ratio
SVM	Support Vector Machine
VPM	Vulnerability Prediction Model
WMC	Weighted Methods per Class

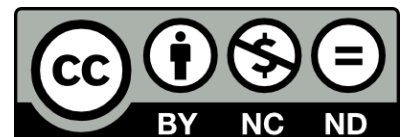
Declaration of Originality

I, Miltiadis Siavvas, declare that this thesis, entitled “*Static Analysis for Facilitating Secure and Reliable Software*” and the work presented in it are my own. I confirm that:

- This work has been done wholly or mainly while in candidature for a research degree at Imperial College of Science, Technology and Medicine.
- Where I have consulted the work of others, this is clearly attributed.
- All the sources and material that are presented in the present thesis and that are not my own are properly referenced and acknowledged.

Copyright

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.



Contents

Abstract	1
1 Introduction	19
1.1 Motivation	19
1.1.1 Software Security	20
1.1.2 Software Reliability	23
1.2 Problem Statement and Expected Impacts	24
1.3 Thesis Contributions	26
1.4 List of Publications	27
1.5 Thesis Outline	28
2 Literature survey	31
2.1 Quantitative Security Assessment	32
2.1.1 Attack Surface	32
2.1.2 Internal Software Security Measurements	33
2.1.3 Quality and Security Assessment Models	35
2.1.4 Open Issues and Contributions	38
2.2 Vulnerability Prediction	41
2.2.1 Software Metrics	41
2.2.2 Text Mining	44
2.2.3 Static Analysis	46

2.2.4	Comparison of Existing Models	47
2.2.5	Open Issues and Contributions	47
2.3	Optimum Checkpoint Recommendation	49
2.3.1	Transaction-oriented Systems and Optimum Checkpoint Interval	50
2.3.2	Long-running Software Applications	52
2.3.3	Open Issues and Contributions	56
3	Software Metrics as Indicators of Specific Vulnerability Types	59
3.1	Experiment Setup	60
3.1.1	Code Base Construction	60
3.1.2	Software Metrics Selection	61
3.1.3	Weakness Categories Construction	63
3.1.4	Correlation Analysis	66
3.2	Results and Discussion	67
3.2.1	Correlation between Software Metrics and Weakness Categories	67
3.2.2	Correlation among Weakness Categories	69
3.3	Summary and Critical Analysis	70
4	Security Assessment Model	73
4.1	Model Description and Construction	75
4.1.1	Model Description	76
4.1.2	Model Construction	85
4.2	Implementation	97
4.3	Model Evaluation and Discussion	100
4.3.1	Comparison with an Independent Security Evaluation Approach	100
4.3.2	Discretion Power of the Model	105
4.3.3	Impact of Software Metrics	115

4.3.4	Class-level Assessment and Vulnerability Prediction	116
4.4	Summary and Critical Analysis	120
5	Optimum Checkpoint Interval: Time Considerations	123
5.1	Indicative Examples	125
5.2	Expected Execution Time of a Program Without and With Checkpoints	126
5.2.1	Expected Execution Time Without Checkpoints	127
5.2.2	Estimating the Failure Probability g	128
5.2.3	Optimum Checkpoints	129
5.2.4	Program with a Long Loop	130
5.2.5	Nested Loops	130
5.3	Numerical Examples	131
5.4	Summary and Critical Analysis	132
6	Conclusion	135
6.1	Thesis Summary and Achievements	135
6.2	Future Work	139
	Bibliography	141
A	Static Analysis	161
A.1	Background Concepts	161
A.2	Static Analysis for Security	163

List of Tables

2.1	Qualitative comparison of existing security assessment models.	40
3.1	Software Metrics	61
3.2	Weakness Categories	64
3.3	Correlations among Software Metrics and Weakness Categories	67
3.4	Correlations among Weakness Categories	69
4.1	The <i>Security Characteristics</i> of the proposed security assessment model. The overall concept of <i>Software Security</i> is hierarchically decomposed into three security characteristics, namely <i>Confidentiality</i> , <i>Integrity</i> , and <i>Availability</i> , which are also known as the CIA triad [163]. The definitions were retrieved from the ISO/IEC 25010 [26] international standard.	78
4.2	The software metrics-based properties of the proposed security assessment model, along with the specific metrics that are used for their quantification. These metrics are retrieved from the Chidamber and Kemerer [153] and QMOOD [78] metric suits, while they are quantified using the CKJM Extended [154] tool. . .	80
4.3	Mapping between PMD rules and CWE entries - Representative Examples . . .	80
4.4	The recommended discrete scoring scheme of the proposed security assessment model. The last column displays the percentage of the applications of the benchmark repository that was used for the calibration of the model that belong in each Security Class.	85
4.5	The final thresholds of the proposed security assessment model.	88

4.6	The general structure of the pairwise comparison (pwc) matrix that is required for calculating the weights of a characteristic C_k of a security model with M properties, namely $P_1, P_2, \dots, P_M$. The value of e_{ij} denotes how stronger (or weaker) the impact of P_i on the C_k is, compared to the corresponding impact of P_j . The pairwise comparison matrix is normally completed with expert judgments.	89
4.7	The impact scores of the model properties on security characteristics (calculated based on CWE).	93
4.8	The weights assigned by the SMARTS/SMARTER approach for a various number of elements (i.e. M). The table is adapted from [166].	94
4.9	The rankings of the model's properties for each characteristic, along with their relative impact scores, retrieved from the SMARTS/SMARTER [166] approach.	95
4.10	The final weights of the model's characteristics as derived from the AHP [162] approach. The weights reflect both the impacts retrieved from the CWE knowledge base, and the expert judgments.	96
4.11	The security evaluation results of the selected open-source software products, as produced by the proposed security assessment model (i.e., SI), the FindBugs static code analyzer (i.e., SAVD), and a security model that completely omits software metrics (i.e., SI'). Their corresponding rankings are also presented.	101
4.12	The list of security issues (i.e., potential vulnerabilities) that the <i>Vulnerability Injector</i> is currently able to inject	107
4.13	The <i>Security Indexes</i> of open-source applications after going through vulnerability injection.	108
4.14	The results (i.e., <i>p-values</i>) of the Shapiro-Wilk Normality Test.	113
4.15	The results (i.e., <i>p-values</i>) of the post-hoc Nemenyi Test.	114

List of Figures

4.1	The overall structure of the proposed security assessment model (SAM)	77
4.2	The general structure of the utility functions of a given security property P . The utility function assigns a continuous value between 0 and 1 to the corresponding property, based on the system-level (i.e., normalized) value m of its associated code-level measure and on a set of data-driven thresholds (i.e., t_l , t_m , t_u).	83
4.3	The system-level values of the security properties of 20 software applications retrieved from the complete benchmark repository used for the calibration of the proposed security assessment model.	87
4.4	CWE provides information about the impact that each weakness has on the main security requirements of software products.	91
4.5	The pairwise comparison matrix used for the calculation of the model's weights that correspond to the characteristic of Confidentiality.	96
4.6	The graphic user interface of the offline version of the proposed security assessment model.	98
4.7	The Security Dashboard of the SDK4ED platform.	99
4.8	The flow chart of the <i>Vulnerability Injector</i>	107
4.9	The boxplots of the <i>Security Indexes</i> of each injection groups.	111
4.10	Distribution of the security scores of the vulnerable and clean software components of the dataset, as produced by (a) the original model, and (b) the recalibrated model.	119
5.1	The additional code that should be inserted for adding an application-level checkpoint in a lengthy loop, using CRAFT (Adapted from [33]).	125

5.2	The additional source code (marked with red color) required for inserting an application-level checkpoint into a loop, using: (a) an extension of the CPPC tool proposed by Losada et al. [122], and (b) the ITALC ALCR library [34].	126
5.3	Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K , for a software application having a small computational loop with $M = 1000$	131
5.4	Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K , for a software application having a medium length computational loop with $M = 10^4$	132
5.5	Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K , for a software application with a large computational loop with $M = 10^5$	133
5.6	Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K , for a software application with a very large computational loop with $M = 10^7$	134

Chapter 1

Introduction

1.1 Motivation

Software is the “lifeblood” of modern society since the majority of critical daily activities (e.g., transactions, communications, etc.) are controlled by some kind of software [1]. Modern software applications are usually large, complex and interconnected through the Internet, while it is common to be available on the Cloud and provide services based on the *Software-as-a-Service (SaaS)* paradigm. Although this makes their features highly accessible and visible to the broader user community, it increases their exposure to malicious individuals, while their accessibility by multiple users may also cause reliability issues. Hence, due to the criticality of the activities that are handled by software products, there is a strong need for dependable software [2]. More specifically, the users wish the available applications to be highly secure, i.e., to withstand malicious attacks, as well as highly reliable, i.e., to tolerate potential failures. Therefore, the *security* and *reliability* of software applications are aspects of major concern for software development enterprises that wish to deliver dependable products and services to their customers. These quality attributes should be taken into account from the early stages of software development.

1.1.1 Software Security

Software security is usually considered an afterthought in the software development lifecycle (SDLC). It is normally added after the implementation of software products both by using mechanisms aiming to prevent malicious individuals from exploiting existing vulnerabilities (e.g., intrusion detection systems), and by fixing reported security issues, a technique commonly known as *penetrate and patch* [3]. However, the increasing number of security incidents reported annually indicates the inability of the traditional defensive mechanisms to fully protect the software products against attacks [4]. In addition, code refactoring, a task required by the *penetrate and patch* technique, has been found to be the main reason for the corruption of existing flawless code, which often leads to the introduction of new vulnerabilities to software systems [5–7]. To this end, software houses have shifted their focus from traditional reactive approaches, towards more proactive techniques for adding security to software products. In particular, emphasis is given on how to build software products that are highly secure from the ground up¹.

A software vulnerability is a weakness in the specification, development, or configuration of software such that its exploitation can violate a security policy [8]. Most of the software vulnerabilities stem from a small number of common programming errors [9, 10]. These errors are introduced by the developers during the coding phase, mainly due to their lack of security expertise [1]. However, it is unrealistic to expect from them to remember thousands of security-related bug patterns and bad practices that they should avoid. Another reason that leads to the introduction of security issues in software products is the accelerated production cycles [11]. Strict deadlines that should be met often enforce developers to focus mainly on the implementation of the predefined functional requirements, neglecting in that way the security of the code they produce [12, 13]. As a result, efficient tools are required to help them avoid the introduction of such security bugs, and therefore write more secure code [12, 13].

Automatic static analysis (ASA) tools have been proven effective in uncovering security-related bugs early enough in the software development process [9, 14]. Their main characteristic is that they are applied directly to the source or compiled code of the system, without requiring its execution [1]. In fact, automatic static analysis is considered an important technique for adding

¹The term “secure software” is used to describe software products that encompass as few vulnerabilities as possible.

security during the software development process. This belief is expressed by several experts in the field of software security (e.g. [9,15]), while well-established secure software development lifecycles (SDLCs), including the well-known Microsoft's SDL [16,17], OWASP's OpenSAAM², and Cigital's Touchpoints [1], propose the adoption of static analysis as the main mechanism for adding security during the coding (i.e., implementation phase) of the SDLC. In addition, ASA is a security activity commonly adopted by major technological firms including Google, Microsoft, Adobe and Intel, as reported by the BSIMM³ initiative.

Moreover, static analysis is believed to be more effective in detecting code-level vulnerabilities compared to other dynamic testing approaches like penetration testing [18] and fuzzing [19], as it is observed to produce significantly fewer false negatives [20]. This can be explained by the fact that software vulnerabilities often exist in states that are hard-to-reach, and that ASA tools can peer into these states more efficiently than dynamic analysis [9]. In addition to this, contrary to dynamic analysis, static analysis does not require an executable version of the source code to be available in order to be applied, allowing its execution from the early stages of the development process, and therefore the early identification of security issues, which is important for secure software development. Hence, based on the above analysis, ASA tools constitute an attractive solution for software development enterprises that wish to consider security from the coding phase of their applications.

Despite the benefits of the ASA tools in the domain of software security, it has been observed that they are underused in practice [21,22]. The main reason for their limited adoption is their poorly presented results [13]. These comprise long lists of raw warnings (i.e., alerts) or absolute values of software metrics, which do not provide real insight to the stakeholders of the software products. A great number of ASA tools have been proposed over the years providing a huge volume of such raw data, which may contain security-relevant information that may be useful for secure software development. Hence, appropriate knowledge extraction tools are needed on top of the raw results produced by ASA tools for facilitating the production of secure software.

To this end, the outputs of the ASA tools can be used for the conduction of quantitative security assessment (QSA). A quantitative expression of software security provides a broader view of the security level of the product under development. The developers and software project

²<http://www.opensamm.org/>

³<https://www.bsimm.com/>

managers can use this information to make informed decisions concerning the implementation of the product at hand. However, despite its importance in secure software development, current literature lacks an operationalized and reliable software security assessment approach [23]. In fact, although a lot of work has been conducted in the field of quality evaluation [24], limited contributions exist, specifically targeting the problem of software security assessment (e.g., [25]). Recently, ISO/IEC 25010 [26] defined, for the first time, security as one of the main characteristics of software quality. This opens a new area of research on software quality evaluation, i.e., for investigating whether the approaches that were originally proposed for software quality evaluation can be applied (and potentially extended) for assessing software security as well, and in which ways.

Vulnerability prediction is another mechanism known to facilitate the production of secure software. The research in the field of vulnerability prediction focuses on examining the ability of several software-related factors (e.g., complexity, coupling, and cohesion software metrics) to indicate the existence of vulnerabilities in software products or components, as well as on the construction of vulnerability prediction models (VPMs) based on these factors [27]. Despite the important contributions that have already been achieved, there are still many open issues that require further research. For instance, the majority of existing research attempts were based on a highly limited number of software applications, which hinders the reliability and the generalizability of their results. In addition, emphasis was given almost exclusively on the ability of software-related factors to indicate the existence of vulnerabilities in general, and not of specific vulnerability types, whereas the existence of potential interdependencies between different types of vulnerabilities has not been studied yet. There are also several previously uninvestigated (or scarcely investigated) factors with potential relevance to software security (e.g., additional software metrics, static analysis alerts, etc.) that deserve individual merit. Finally, another issue is that existing VPMs are based on Machine Learning (ML) algorithms, which significantly hinders their transparency.

In this respect, an interesting research topic would be to examine the ability of software-related factors (including previously uninvestigated or scarcely investigated factors) to indicate the existence of specific vulnerability types in software products, as well as to investigate whether potential interdependencies exist between different vulnerability types. Another interesting topic would also be to investigate the feasibility of constructing a highly transparent (i.e.,

non-ML-based) VPM with sufficient discriminative power and practicality.

1.1.2 Software Reliability

Reliability is an important requirement for modern software products, especially for long-running software applications that perform computationally expensive tasks. In such applications, a single failure may lead to the re-execution of a large number of operations, leading to significant overheads in important Quality-of-Service (QoS) parameters, including performance and energy consumption. In order to address this issue, several fault tolerance mechanisms have been proposed over the years [28–30], with the Checkpoint and Restart (CR) [31] mechanism to be the most widely used. In brief, the CR mechanism periodically keeps a safe copy of the program execution state (i.e., a checkpoint), and it uses this copy for restarting the program in case of a failure. Hence, in case of a failure, the program resumes its execution from the most recent checkpoint, avoiding in that way excessive re-executions.

Among the existing checkpointing strategies, the Application-level Checkpoint and Restart (ALCR) [32] is considered the most efficient, since it leaves the smallest memory footprint [33, 34]. ALCR is a proactive mechanism for adding fault tolerance to software applications since it allows the developers to programmatically insert the checkpoints directly into their source code. Hence, it is widely adopted in practice, as it enables the development of software applications that are fault tolerant, and therefore reliable from the beginning. However, since it necessitates source code modifications, it requires significant development effort for the insertion of the checkpoints into the applications. Significant expertise is also required for optimally selecting their locations and checkpoint intervals.

Existing ALCR tools and libraries (e.g., [33, 34]) manage to reduce the manual effort required by the developers, by (i) automatically identifying (normally through static analysis) judicious locations in which checkpoints should be inserted (in fact, long loops), and by (ii) automating the insertion of the checkpoints into the identified locations. However, their major shortcoming is that they do not provide recommendations regarding the optimum checkpoint interval. Hence, the checkpoint interval is selected by the developers usually in an arbitrary manner. However, since the ALCR mechanism may introduce overheads itself (due to the insertion of additional code), suboptimal values of checkpoint interval may affect critical QoS parameters

of the software applications. For instance, in a software application that failures are rare occurrences and the cost of checkpointing is high, taking frequent checkpoints may potentially lead to an increase in the average execution time of the program. Although the impact of the selection of the checkpoint interval on other qualities has been extensively studied for the case of transaction-oriented systems [35–37], no similar attempts exist for modern software applications that adopt the ALCR mechanism.

Hence, an interesting research topic would be to formally assess the impact that the adoption of the ALCR mechanism, and, in turn, the selection of the inter-checkpoint interval may have on important QoS parameters of an application, like performance (i.e., execution time). Based on this, another interesting topic would be to investigate the possibility of proposing a method for determining the optimum checkpoint interval, i.e., the interval between two successive checkpoints that optimizes one or more QoS parameters.

1.2 Problem Statement and Expected Impacts

The purpose of the present thesis is to investigate novel techniques and tools that are directly or indirectly based on static analysis for facilitating the development of secure and reliable software. In particular, its main objective is to advance the current state of the art of the three individual subtopics that were described in Section 1.1, namely: (i) quantitative security assessment, (ii) vulnerability prediction, and (iii) optimum checkpoint recommendation, by attempting to address existing open issues that have not been sufficiently tackled yet.

As far as software security is concerned, the main goal of the present thesis is to develop methods able to extract security-relevant knowledge from the results produced by static analysis tools, in order to support secure software development. More specifically, the problem that we attempt to address with respect to software security can be stated as follows: Given the raw results produced by a set of static analysis tools for a specific software product, the question is (i) how these results can be aggregated in a meaningful way for conducting quantitative security assessment, and (ii) whether these results can be used as reliable indicators of vulnerabilities. The quantitative security assessment is expected to provide a broader high-level view of the product’s security level to its developers and project managers, whereas the vulnerability indicators are expected to assist them in prioritizing their testing and inspection efforts, by

allocating limited test resources to potentially vulnerable areas.

More specifically, within the context of the present thesis the development of an assessment model able to quantify the internal security level of software products based on low-level results produced by static analysis is attempted. The ability of the proposed model to indicate the existence of vulnerabilities in software components is also investigated, with the purpose evaluate the feasibility of the proposed model to be used as the basis for the construction of a highly transparent VPM. Since the model is based on static analysis, a positive answer to this question would also verify the capacity of statically collected factors (i.e., static analysis alerts and software metrics) to indicate the existence of software vulnerabilities in general. In addition, the ability of software metrics (including previously uninvestigated metrics) to indicate the existence specific types of vulnerabilities is examined, whereas special emphasis is given on the potential inter-relationships among different types of security issues.

As far as software reliability is concerned, emphasis is given on the selection of the optimum checkpoint interval of application-level checkpoints. As already mentioned, although several ALCR libraries and tools exist for facilitating the insertion of checkpoints in judicious locations of the source code (usually long-running loops), none of them provide recommendations regarding the checkpoint interval of the inserted checkpoints, while no research attempts exist focusing on the impact of the checkpoint interval on various QoS parameters. Hence, within the context of the present thesis, we attempt to provide a mathematical model able to assess the impact that the inter-checkpoint interval may have on the performance (i.e., execution time) of software applications, which is an important QoS parameter. In addition, based on this model we provide a numerical approach for calculating the optimum checkpoint interval, i.e., the interval between two successive checkpoints that minimizes the average execution time. The former is expected to highlight the consequences that the arbitrary selection of the checkpoint interval may have on the performance (i.e., execution time) of software applications, while the latter is expected to help developers make more informed decisions about the inter-checkpoint interval of application-level checkpoints.

Finally, the present thesis is also expected to provide some indirect contributions. In particular, for the construction and the evaluation of the aforementioned methods, a large number of real-world software applications were collected and analyzed. The produced datasets have been made publicly available in order to facilitate further research endeavors [38]. For instance,

these data could form the basis for the conduction of relevant empirical studies, like those that we have conducted in [39] and [40].

1.3 Thesis Contributions

The main contributions of the present thesis are listed below:

- An empirical evaluation of the ability of software metrics to indicate the existence of specific types of vulnerabilities in software products was performed. Previously uninvestigated metrics were considered, while the analysis was based on a much larger repository of real-world software applications. The existence of potential interdependencies among different vulnerability types was also examined.
- A hierarchical *Security Assessment Model (SAM)* has been proposed that allows the quantitative security evaluation of software products based on static analysis alerts and software metrics. The proposed model is fully-automated, allowing its regular application during the development cycle, while it is also operationalized in the form of individual tools, which enable its adoption in practice.
- A mathematical model has been proposed for determining the optimum checkpoint interval, i.e., the checkpoint interval between successive application-level checkpoints that minimizes the average execution time of an application that adopts the ALCR fault tolerance mechanism.

Through the research work conducted for the purposes of the present thesis, some additional contributions were also made, which are worth mentioning. These additional contributions are listed below:

- For the construction of the proposed SAM, a novel weights elicitation approach was proposed, which allows the derivation of a reliable set of weights that reflect the well-established knowledge expressed by the *Common Weakness Enumeration (CWE)* [41]. This approach is based on popular decision-making techniques.

- An investigation of the ability of the proposed SAM to discriminate between vulnerable and clean software components was conducted. This investigation is important for determining whether the proposed SAM could be used as the basis for the construction of VPMs.
- Based on the proposed mathematical model, an evaluation of the impact that the value of the checkpoint interval may have on the performance (i.e., average execution time) of software applications that adopt the ALCR mechanism was performed.

1.4 List of Publications

Parts of this thesis have been published or submitted for publication in a number of peer-reviewed conferences and journals. A list of these articles can be found below:

Conferences

- Siavvas M., Kehagias D., Tzovaras D., “A preliminary study on the relationship between software metrics and specific vulnerability types,” International Conference on Computational Science and Computational Intelligence, (2017).
- Siavvas M., Gelenbe E., Kehagias D., Tzovaras D., “Static analysis-based approaches for Secure Software Development,” Security in Computer and Information Sciences, Communications in Computer and Information Science, (2018).
- Siavvas M., Gelenbe E., “Optimum interval for application-level checkpoints,” 6th IEEE International Conference on Cyber Security and Cloud Computing, (2019), *accepted*.

Journals

- Siavvas M., Kehagias D., Tzovaras D., Gelenbe E., “A hierarchical model for quantifying software security based on static analysis alerts and software metrics,” Journal of Systems and Software, (2019), *submitted*.

- Siavvas M., Gelenbe E., “Optimum checkpoints in programs with loops,” Journal of Simulation Theory and Practice, (2019), *accepted*.

Other publications that have been produced and are related to the work presented in the present thesis are listed below. It should be noted that for reasons of brevity the contents of these papers are not described in detail in the rest of the present thesis.

- Siavvas M., Jankovic M., Kehagias D., Tzovaras D., “Is popularity an indicator of software security?,” 9th IEEE Conference on Intelligent Systems, (2018).
- Siavvas M., Tsoukalas D., Jankovic M., Kehagias D., Chatzigeorgiou A., Tzovaras D., Anicic N., Gelenbe E., “An empirical evaluation of the relationship between technical debt and software security,” 9th International Conference on Information Society and Technology, (2019), *accepted*.

1.5 Thesis Outline

The rest of the present thesis is organized as follows. Chapter 2 provides a comprehensive literature review of existing work in important subfields that are relevant to the broader research topic of the present work. The most representative contributions in these subfields are presented and critically analyzed, whereas open issues that remain unresolved are identified and reported. This literature survey facilitates the identification of those research topics that have potential for further investigation, motivating the work proposed in the current thesis.

In Chapter 3 an empirical evaluation of the ability of statically collected software metrics (including previously uninvestigated metrics) to indicate the existence of security issues in software products is provided. More specifically, the capacity of software metrics to discriminate between different types of vulnerabilities is studied, along with an examination of potential inter-relationships among different vulnerability types.

In Chapter 4, the proposed *Security Assessment Model (SAM)* that is able to quantify the internal security level of software applications based on static analysis alerts and software metrics is presented. A detailed evaluation of the proposed model on a large volume of empirical

data is also provided, along with an examination of its ability to indicate the existence of vulnerabilities in individual software components (i.e. classes).

In Chapter 5, a mathematical model that allows the calculation of the checkpoint interval of application-level checkpoints that minimizes the average execution time of a software program is presented. Based on the proposed mathematical model, the impact that the selection of the inter-checkpoint interval may have on the performance of software application is evaluated and illustrated through a number of numerical examples.

Chapter 6 summarizes the work conducted, discusses directions for future work and concludes the thesis. Finally, an Appendix section is also provided, in which important background concepts are presented in order to familiarize the reader with the concept of static analysis, which is relevant to the present research.

Chapter 2

Literature survey

The problem description, as stated in Chapter 1, implies that three different research subfields need to be studied. These are: (i) *quantitative security assessment*, which refers to the derivation of a quantitative expression of software security; (ii) *vulnerability prediction*, which refers to the identification of potential indicators of security issues in software products; and (iii) *optimum checkpoint recommendation*, which focuses on optimizing the insertion of checkpoints in software applications. In the present section, we provide a review of the most significant contributions that have been conducted in these three subfields, while specific emphasis is given on their open issues that have not been sufficiently addressed yet.

The identified open issues helped us define the main objectives of the present thesis, and also decide the specific problems that the present research attempts to address. In particular, the research endeavors presented in Chapters 3, 4, and 5 manage to address important open issues that are reported in this chapter. In addition, this chapter can act as a reference for researchers that wish to contribute in the aforementioned subfields, in order to gain a solid understanding of the existing solutions and their open issues that require further research.

It should be noted that due to the fact that static analysis is a basic concept of the present research since it is directly or indirectly related to the aforementioned areas, a description of its background concepts and the related work in its field would be valuable. However, for reasons of brevity, this analysis is provided in the appendices of the present thesis.

Part of this chapter has appeared in the proceedings of the Security Workshop of the 32nd International Symposium on Computer and Information Sciences [42]

2.1 Quantitative Security Assessment

Quantitative security assessment (QSA) is the process of providing a quantitative expression of software security. This quantitative expression is more intuitive and easily understandable even by stakeholders with little or no technical knowledge, and therefore it can be used to facilitate decision making during the secure software development. QSA is highly important during the secure SDLC as a common belief in software engineering is that *“you cannot control something that you cannot measure”* [43]. In fact, security metrics are valuable for the developers, who want to quantify the security improvements made by their corrective actions [44]. QSA is a scarcely investigated field, which only recently has started gaining the attention of the research community, as a result of the overall shift of software industry towards secure software development.

2.1.1 Attack Surface

The attack surface is the most complete and well-accepted measure for quantifying the security level of software products that has been proposed in the literature until today. Intuitively, a system’s attack surface is a set of ways in which an adversary can enter the system and potentially cause damage [44]. The notion was originally introduced by Michael Howard, who also proposed a method for its quantification, called Relative Attack Surface Quotient (RASQ) [45]. RASQ is based on the identification of a set of attack vectors (i.e., features of the product that can be used in an attack), and subsequently on the assignment of weights that reflect their attackability (i.e., likelihood of a feature being used in an attack). According to Howard’s approach, the attack surface of the product under assessment is the weighted count of the identified attack vectors. An empirical evaluation of RASQ on several versions of Windows [45] and Linux [46] operating systems, highlighted the ability of the proposed measurement to reflect the perceived security. However, the main shortcoming of RASQ is that it requires significant security expertise both for the selection of the appropriate attack vectors and for the assignment of weights to these vectors, rendering its adoption by non-security experts highly difficult [44].

In an attempt to address these shortcomings Manahdata et al. [44] proposed an extension of RASQ, called Attack Surface Metric (ASM). In this approach, the attack surface of the

software product is defined based on the resources that are used in attacks, namely the system's methods (i.e., entry and exit points), channels, and data. Some resources are more likely to be exploited than others and, thus, they contribute more to the system's attack surface. The notion of damage potential-effort (DP-E) ratio is used to estimate these contributions. The main advantage of ASM over RASQ is that the calculation of the former does not require security expertise, and therefore it can be applied regularly by software developers during the overall SDLC. In fact, ASM is the most widely-used metric for quantifying the attack surface of software products.

Several improvements of the notion of the attack surface, and especially of the ASM have been proposed in the related literature until today [47]. For instance, in [48] the traditional ASM metric is extended to consider the characteristics of privacy and dependability, in addition to security. The proposed approach also takes into consideration the effectiveness of the protection mechanisms that are implemented by a software system, providing a more complete estimation of the system's protection level compared to the original metric, which estimates only its attackability (i.e., susceptibility to attacks). In addition, in [49] another extension of the ASM is proposed in order for the internal structure of the software products to be considered during the calculation of their attack surface. The authors exploit the mechanisms of call graphs and random walks in order to provide function- and file-level attack surface metrics, allowing in that way finer-grained risk assessment. Call graphs are used to represent the internal structure of a software product, while random walks are used to simulate the attacker behavior when exploring the system for vulnerabilities. Although the proposed approach considers more internal information of the software products compared to the original metric, it still neglects significant project-specific low-level characteristics that influence software security (e.g., vulnerabilities).

2.1.2 Internal Software Security Measurements

Despite its importance for risk mitigation during the SDLC [44, 45], the main shortcoming of the attack surface is that it neglects the innate characteristics of software products that are known to influence software security (e.g., vulnerabilities). In fact, it is an external security measure that quantifies only the attackability of a software product, i.e., the likelihood of vulnerability exploitation. However, the value of a product's attack surface is independent of the total number of the vulnerabilities that it contains [44]. Hence, in order to achieve a

more complete security assessment, attack surface metrics should be used in conjunction with measurements that take into account the internal characteristics of software products [44], such as (i) security-relevant software metrics, and (ii) vulnerability-based security measurements.

While the ability of common software metrics to indicate security risks has extensively been examined [50–59] (see Section 2.2.1), only a limited number of custom security metrics can be found in the related literature. For instance, Chowdhury et al. [60] proposed three code-level security measurements, namely *stall ratio (SR)*, *coupling code propagation (CCP)*, and *critical element ratio (CER)*, which are able to quantify specific quality properties of code fragments that have an obvious impact on software security. Similarly to [60], Alshamari et al. [61] proposed a set of seven design-level metrics for quantifying the security level of object-oriented classes. These metrics are able to measure the cohesion and encapsulation of a given class, from the viewpoint of potential information loss. The same authors proposed a new set of measurements for assessing information-flow security of object oriented designs [62]. Contrary to their previous work [61], the new set of metrics is able to assess the security based on the entire class hierarchy instead of a single class diagram. Hence, the proposed approach can be used for comparing the security of various alternative designs. Finally, Abdulrazeg et al. [63] introduced a set of security metrics for quantifying and improving the misuse case model. The proposed measurements are applied during the requirement phase, which allows the early identification and mitigation of vulnerabilities.

The major shortcoming of these metrics is that they are able to assess only specific aspects of software security, while they are also custom metrics that have not been extensively evaluated for their ability to reflect security concerns. In addition, they are not operationalized in the form of a tool, and therefore they cannot be used in practice.

As far as the vulnerability-based security measurements are concerned, the *vulnerability density* metric is the most representative measure of this category. It was introduced by Alhazmi et al. [64] as a way to assess the overall security level of software products based on their reported vulnerabilities. The *vulnerability density* is defined as the ratio of the total number of the vulnerabilities that a product contains to the product size expressed in thousand lines of code. The static analysis vulnerability density (SAVD) [65] is an alteration of the original *vulnerability density* metric that is calculated based on the security-related static analysis alerts of a software product, instead of its reported vulnerabilities. SAVD has been widely used in

the related literature as a security indicator (e.g. [65]), while it has been also used as the basis for the construction of vulnerability predictors (e.g., [66,67]).

Based on SAVD, Walden et al. [68] proposed SAVI (Static Analysis Vulnerability Indicator) for evaluating the security risk of web applications. In fact, SAVI corresponds to an estimation of the total number of the actual vulnerabilities that a web application contains, based on the security-related static analysis alerts produced by Fortify SCA [69], a widely-used commercial static code analyzer. The evaluation of the metric on five popular web applications revealed that the proposed metric can be used effectively for ranking web applications based on their security risk. Finally, Wang et al. [70] extended the notion of *vulnerability density*, by taking into account the severity of the identified vulnerabilities during the metric calculation. However, due to the fact that the severities should be calculated based on the CVSS Base Scores [70,71] of the reported vulnerabilities, their approach require significant manual effort and, thus, it is not easily applicable in practice.

2.1.3 Quality and Security Assessment Models

As observed by the previous analysis, existing measurement approaches are not enough for measuring the overall security level of a software product. They usually measure a specific security aspect (e.g., [60,61]), or they try to quantify the overall security level of the software product, but by considering limited characteristics of the system under evaluation (e.g., [64]). Hence, there is a strong need for a sophisticated approach, i.e., a model, able to combine multiple security metrics in order to provide a reliable indicator of the overall software security. A security model is a formal representation of software security, with the main objective to provide a quantifiable expression of software security by combining several heterogeneous security metrics [72]. The combination of different security metrics allows the model to capture different facets of software security, providing in that way a more reliable and complete assessment of a product's security level. Software security assessment is commonly treated as a subfield of software quality evaluation [73], chiefly due to the fact that software security is one of the main quality characteristics of software products [26]. Hence, security models could potentially be derived based on quality models.

According to [74], a quality model is a model with the objective to describe, assess, and/or

predict quality. A large number of quality models has been proposed in the literature until today [24]. The first known quality models are dated back in the late 1970s when Boehm et al. [75] and MacCall et al. [76] tried to define the complex notion of software quality by hierarchically decomposing it into a set of more tangible characteristics and sub-characteristics. Although they provided a solid definition of software quality, they did not propose a way for reliably assessing these characteristics and, in turn, the overall quality. Hence, the majority of the next research attempts in the field focused mainly on how to quantify these attributes. The most notable attempts towards this direction were made by Dromey [77] and later by Banisya and Davis [78], who formally described how these high-level quality attributes can be measured from low-level properties, which can be directly quantified through software metrics and static analysis. All these research attempts are highly important as they form the basis upon which the majority of the recent quality models were built.

The SIG Maintainability Model [79] and the Quamoco tool chain [80, 81] constitute the most significant contributions so far in the field of quantitative quality assessment based on static analysis. The SIG Maintainability Model [79] is a quality assessment model that evaluates the maintainability of software products. The model decomposes the characteristic of maintainability into a set of sub-characteristics, which are further decomposed into a set of properties. These properties are directly quantifiable from the source code of the product through static analysis. Based on the absolute values of the properties and a set of thresholds, a *quality profile* is assigned to each property. These profiles are then aggregated and the overall quality score of the system is obtained. In order to enhance the assessment's objectivity, the thresholds used for the evaluation of the model's properties are derived via benchmarking [82].

The Quamoco tool chain [80, 81] follows the same reasoning with the SIG Maintainability Model [79], with the only difference that it allows the stakeholders to define their own hierarchical quality models. Therefore, the produced models are not restricted to a specific quality attribute (e.g., Maintainability), and thus more generic models can be produced. Similarly, in a recent attempt, Siavvas et al. [83] introduced QATCH (Quality Assessment Tool Chain), a framework that allows the derivation of custom hierarchical quality models based on static analysis and benchmarking (similarly to [79–81]). Its main novelty is that it also provides multi criteria decision-making techniques for the derivation of the model's weights from expert judgments. This allows the produced weights to better reflect the experts' opinions, leading to

the production of more reliable quality models.

The literature about adapting the quality models to the case of quantitative security assessment is highly limited. Although several quality models have been proposed for evaluating specific quality characteristics like Maintainability (e.g., [79,84,85]) and Reliability (e.g., [86,87]), limited contributions can be found regarding software security. This can be explained by the fact that only recently security was officially recognized as one of the main software product quality characteristics (ISO/IEC 25010 [26]), opening in that way a new area of research on whether existing quality models can be extended to support security evaluation as well.

To this end, towards an attempt to expand the notions originally proposed for software quality evaluation into the security realm, a small number of security models has already been proposed. More specifically, Sen-Tarng Lai [88] proposed an Analyzer-based Software Security Measurement (ASSM) model that calculates the overall security level of software products, based on a set of custom metrics collected through static and dynamic analysis. The model calculates these metrics for a given software product and combines their normalized values using the dynamic linear combination model [89]. In fact, the weighted average of the normalized metrics is used to obtain the overall security score of the evaluated product. The regular application of the model allows the identification and, in turn, elimination of flaws that reside in the system. In [90], a hierarchical model for assessing the security of object-oriented programs from low-level metrics is proposed. The model starts from the calculation of class-level custom software metrics that quantify properties related to the potential flow of classified data between classes [61,62], and combines them in order to quantify higher level security attributes. The entire program security (i.e., confidentiality) is summarized as a single security score, allowing the direct comparison of different software products with respect to their security.

Although these models effectively produce a high-level security indicator by combining low-level security metrics, they are characterized by a set of shortcomings. Firstly, their design parameters (e.g., weights, properties, etc.) are arbitrarily selected (i.e., their selection is not based on a formal model e.g., ISO/IEC 25010 [26]), which puts under question the reliability of the overall assessment. In addition, the security assessment is based on a set of custom metrics that have not been extensively studied for their ability to indicate security risks, while significant manual effort is required for their calculation.

Towards an attempt for a more reliable security assessment, Xu et al. [25], proposed a security

assessment model based on ISO/IEC 25010 [26]. More specifically, the model decomposes the characteristic of security into a set of sub-characteristics (as defined by [26]), which are further sub-divided into a set of security-related properties. One or more reviewers need to examine the source code of the system and evaluate each property by assigning a continuous value (i.e., rating) between 0 and 1. The ratings are then aggregated to obtain the overall security score of the product under assessment. Similarly to [25], Colombo et al. [91] proposed a hierarchical model for measuring access security. Based on ISO/IEC 25010 [26], the model decomposes software security into a set of more tangible characteristics, namely Authentication, Confidentiality, Integrity, and Non-repudiation, which are further decomposed into a set of security properties. These metrics have to be evaluated by security experts, after manually inspecting the source code of the product under evaluation. For this purpose, a multi criteria decision-making technique is employed in order to map expert judgments to security attribute rankings and, in turn, for the final calculation of the overall security score of the product.

Although both models are based on an international standard for the definition of software security, they are characterized by a set of shortcomings. Firstly, they are not automated as their properties need to be manually evaluated, and therefore they cannot be applied regularly during the SDLC. In addition to this, the overall assessment is highly subjective due to the fact that the low-level properties of the models are evaluated based on expert judgments.

Finally, in a relatively recent work, Zafar et al. [92] proposed a way for constructing hierarchical security assessment models based on the Dromey's [77] quality model. In fact, they attempted to extend Dromey's notions that were initially proposed for software quality, into the security realm. Although they provided a formal description on how low-level security properties can be mapped to high-level security attributes, they did not propose an operationalized assessment model that can be used in practice.

2.1.4 Open Issues and Contributions

From the above analysis, it is obvious that quantitative security assessment requires a sophisticated model able to quantify the internal security of software products by combining heterogeneous low-level security-relevant metrics [72]. However, current literature lacks an effective model for measuring the internal security risk of software products. This observation,

which is also supported by a recent survey [23], can be explained by the fact that the field of software security evaluation is a relatively new area of research, which has recently started gaining attention due to the overall shift of the software industry towards secure software development [93, 94]. In fact, although a small number of security models has already been proposed [25, 88, 90–92], none of them has managed to achieve sufficient level of *practicality* and *reliability*. To better illustrate our findings, and in order to facilitate the qualitative comparison of existing models, similarly to [79], we define a set of requirements that a model should satisfy, in order to be considered both *practical* and *reliable*. These requirements are described below:

- **High-level overview:** The model should provide a single quantitative value that reflects the internal security of software products.
- **Operationalization:** The model should be operationalized, i.e., implemented in the form of a tool, and not only a theoretical conceptualization.
- **Automation:** The model should be fully-automated, in order to be used regularly during the overall development process, without significantly affecting the developers' workflows.
- **Standardization:** The model should encapsulate formal concepts that are expressed by international standards (e.g., ISO/IEC 25010 [26]).
- **Objectivity:** The model should provide sufficiently objective security assessments, considering minor (to no) subjective information. In fact, it should avoid being based on (i) custom metrics, (ii) expert judgments, and (iii) arbitrary values or assumptions. In case that subjective information is considered by the model, an evaluation of its impact on the model's objectivity should be at least provided.

The first three requirements define the practicality of the model, while the latter two its reliability (i.e., trustworthiness).

A direct comparison of existing models is considered valuable in order to help the reader understand their major weaknesses, as well as the main contributions of the present work. This comparison is provided in Table 2.1. In fact, the above-mentioned requirements act as the main criteria (i.e., axes) for the comparison.

Table 2.1: Qualitative comparison of existing security assessment models.

Requirements	Sen-Tarng Lai [88]	Alshammari et al. [90]	Xu et al. [25]	Colombo et al. [91]	Zafar et al. [92]	Proposed Model
High-level overview	✓	✓	✓	✓	✓	✓
Operationalization	×	×	×	×	×	✓
Automation	✓	✓	×	×	×	✓
Standardization	×	×	✓	✓	✓	✓
Objectivity	×	×	×	×	×	✓

As can be seen by Table 2.1, none of the existing models manages to satisfy all the defined requirements. An interesting observation is that none of the existing models is operationalized in the form of a tool, and thus they cannot be used in practice for industrial purposes or further research. Another interesting observation is that none of them is sufficiently reliable. In fact, these models are usually based on subjective information, such as expert judgments and arbitrarily selected parameters, while they also lack sufficient empirical evaluation. Their evaluation is based on simple case studies conducted on a limited number of software applications, avoiding comparisons with existing security evaluation approaches. Although these case studies are effective in demonstrating the usefulness of the proposed models, they do not provide sufficient evidence for their correctness. The lack of empirical evaluation is the main shortcoming of the vast majority of existing security metrics [72].

To this end, as can be seen by Table 2.1, the present research attempts to propose a security assessment model that sufficiently satisfies all the requirements defined in this section. More specifically, the present thesis focuses on the development of an operationalized and fully-automated security assessment model that allows the quantitative security evaluation of software products based on low-level security indicators (i.e., security-related static analysis alerts and statically collected software metrics). Fundamental concepts from the field of software quality evaluation are leveraged and extended into the security realm. Contrary to previous research endeavors in the field of software security assessment, specific emphasis is given on the elaborate empirical evaluation of the proposed model on a large volume of real-world data.

2.2 Vulnerability Prediction

Vulnerability prediction is a relatively new area of research in the field of software security that focuses on predicting the existence of security issues (i.e., vulnerabilities) in software products or components. In fact, the research in the field of vulnerability prediction aims at (i) identifying software-related factors (e.g., software metrics) that can potentially indicate security risks in software (e.g., vulnerabilities), and subsequently at (ii) building vulnerability prediction models (VPMs) based on the identified security indicators. VPMs are normally built based on machine learning (ML) techniques that use these factors as inputs, to discriminate between vulnerable and neutral components or to predict the number of the vulnerabilities that potentially reside in software. As stated in [27], the majority of the research attempts in the field of vulnerability prediction focus on the ability of software metrics [50–59] and text mining features [95] to indicate the existence of software vulnerabilities, whereas limited attention has been given recently on static analysis alerts [96]. A critical analysis of the most important research attempts is provided in the rest of this section. It should be noted that the present analysis is of specific interest for the broader research topic since the identified vulnerability predictors are sufficient security indicators that can also be used for software security assessment.

2.2.1 Software Metrics

Software metrics have been widely used in the literature for predicting fault-prone software components [97–101]. Since software vulnerabilities and defects are dissimilar groups [73, 102], the efficacy of software metrics to indicate the existence of software vulnerabilities has been studied individually. Shin and Williams [50, 51], were the first to investigate the ability of software metrics, particularly complexity metrics, to predict vulnerabilities in software products. More specifically, in [50] they performed an empirical analysis based on the reported vulnerabilities of Mozilla Javascript Engine and on nine complexity metrics that were calculated at function-level of granularity. The results of their analysis revealed a weak but statistically significant correlation between software complexity metrics and the number of vulnerabilities, suggesting that these factors could potentially be used as vulnerability indicators. Based on this observation in [51] the same authors examined the ability of complexity metrics to discrim-

inate between vulnerable and non-vulnerable (i.e., neutral) components retrieved from Mozilla Javascript Engine. Discriminant analysis was employed while several regression models were built for this purpose. The results of the analysis provide support to their previous observation that complexity metrics can be used as weak indicators of software vulnerabilities.

Based on the same code base (i.e., Mozilla Javascript Engine), Nguyen and Tran [52] evaluated the ability of a new set of semantic complexity metrics (extracted from the dependency graphs of the product software components) to predict vulnerabilities. The models that were built using semantic complexity demonstrated on average a better predictive performance compared to the best models of [51]. In [53] an empirical study grounded on two large open-source software products, namely Mozilla Firefox and Whiteshark, was conducted with the purpose to examine the ability of execution complexity metrics (i.e., complexity metrics that are collected during the code execution) to discriminate between vulnerable and non-vulnerable code locations. The results of the study revealed that execution complexity metrics are able to discriminate between vulnerable and non-vulnerable code locations with similar discriminative power to commonly used statically-collected complexity metrics.

The main purpose of the previous studies was to empirically evaluate the experts' opinion that software complexity is the enemy of software security. The weak relationship that was generally observed between complexity and vulnerabilities led to the need for incorporating additional metrics in vulnerability prediction.

Towards this end, Chowdhury and Zimmerman [54], based on 52 releases of Mozilla Firefox, highlighted the ability of complexity, coupling, and cohesion (CCC) metrics to indicate the existence of vulnerabilities in software files. In particular, the correlation analysis that they applied revealed that CCC metrics are closely correlated to the number of reported vulnerabilities at a statistically significant level. Based on this observation, the same authors proposed a framework for the automatic prediction of vulnerable files based on CCC metrics [55]. The VPMs that were built demonstrated high accuracy and tolerable false positive rate. Similarly to these attempts, Shin et al. [56] examined the ability of complexity, code churn, and developer activity to discriminate between vulnerable and neutral files in two widely-used software products, namely Mozilla Firefox and Red Hat Linux Kernel. The results of their analysis suggest that the selected metrics have adequate discriminative power, and therefore they may be used as sufficient indicators of vulnerabilities in software files. Several logistic regression models were

built using these metrics, demonstrating sufficient file-level predictive performance. However, it should be noted that the previous attempts [54–56] provide predictions at a very high level of granularity (i.e., file-level granularity), which obviously hinders the practicality of the produced models.

Previous studies were based on a highly limited number of products, which hinders the reliability and generalizability of their results. Hence, several research endeavors focused on investigating the generalizability of these findings using more and previously unused software applications. For instance, in [65] 14 widely-used open-source web applications written in PHP programming language were analyzed by calculating a wide range of system-level metrics and their static analysis vulnerability densities. The results of the analysis showed that there is a statistically significant but weak correlation between the vulnerability densities and the calculated metrics of the selected software products, which supports the findings of the previous studies. Similarly, in [59], an empirical analysis was conducted based on function-level vulnerabilities retrieved from the security patches of five widely-used open-source software products, namely Linux Kernel, Mozilla Firefox, Xen Hypervisor, HTTPD, and Glibc, to find relationships between CCC metrics and vulnerabilities. The results indicate that software metrics are able to discriminate between vulnerable and non-vulnerable functions, which underpins the observations made by previous research attempts.

In a recent work, Medeiros et al. [103] extended the work presented in [59], by building several machine learning models based on different subsets of CCC metrics produced through a feature selection algorithm. The results of their study, which was based on the same dataset that was used in [59], revealed a strong correlation between several project-level metrics and the number of reported vulnerabilities, while the produced predictors demonstrated sufficient predictive performance. Both of these observations support the belief that software metrics can be used as vulnerability indicators. In [104], several size and complexity metrics were calculated for three open-source software applications developed by Mozilla, namely SeaMonkey, Firefox, and Thunderbird and were used as the basis for constructing vulnerability predictors. Their results verified the ability of software metrics to be used as indicators of vulnerabilities, whereas they also suggest that deep learning can lead to better predictions compared to conventional ML algorithms.

Finally, the predictive performance of software metrics in cross-project prediction has been also

studied. In fact, Moshtari et al. [57], examined and highlighted the ability of software complexity to predict vulnerabilities between software products (i.e., cross-project prediction), based on five open-source software applications, namely Mozilla Firefox, Linux Kernel, Apache Tomcat, Eclipse, and Open SCADA. Similarly, in a recent study [58], the predictive power of complexity and coupling in cross-project prediction was compared. The results revealed that complexity metrics had better predictive power than coupling metrics in cross-project prediction, while the combination of traditional complexity measurements with a newly proposed set of coupling metrics led to an improvement in the recall of the best complexity-based VPM that was built in this work.

2.2.2 Text Mining

Apart from software metrics that have received much attention in the field of vulnerability prediction, VPMs using text mining have also demonstrated highly promising results. In this approach, the source code of the software artifacts is parsed and represented as a set of tokens (i.e., keywords). Subsequently, these tokens are intuitively combined and used to train vulnerability predictors. Neuhaus et al. [105] were the first to adopt a form of text mining for vulnerability prediction. They proposed Vulture, a VPM that predicts vulnerabilities in software components based on their import statements and function calls. In particular, the source code is parsed in order to retrieve all the import statements and function calls that are more frequently present in vulnerable components, and this information is then used for vulnerability prediction. An empirical evaluation of the model on Mozilla Firefox and Thunderbird revealed that the proposed VPM was able to predict half of all the existing vulnerable components, and about one third of all the predictions were correct. Vulture is also the first known VPM that can be found in the related literature.

A more complete text mining-based prediction approach was introduced by Hovsepyan et al. [95]. According to their technique, each software component is characterized as a series of text terms extracted from their source code along with their associated frequencies, which are then used to forecast whether each component is likely to contain vulnerabilities. An empirical evaluation on 19 versions of a large-scale Android application, revealed that their technique may be promising for vulnerability prediction, as the produced predictors achieved sufficient precision (85% on average) and recall (87% on average). Based on these preliminary results,

the same authors conducted a more elaborate empirical study to investigate the validity of their approach [106]. In particular, several VPMs using Nave Bayes and Random Forest algorithms were constructed and evaluated on a code base of 20 large-scale Android applications. The evaluation results revealed that the predictive power of the proposed models is equal or even superior to what is achieved by state-of-the-art VPMs, which indicates that text mining can be used for the construction of satisfactory VPMs. However, the produced models performed poorly in cross-project prediction, which can be explained by the fact that their predictions are based on text terms, which are highly project-specific features.

Pang et al. [107] proposed an improvement of the aforementioned technique [95, 106], by employing N-Gram analysis. According to their proposal, continuous sequences of tokens should be used instead of raw text features for predicting vulnerable components. An empirical evaluation based on four android applications (retrieved from the same code base with [106]), revealed that SVM predictors built based on N-Gram technique were able to predict vulnerable components with high accuracy, precision and recall. In a recent replication of their study [108], the same authors observed that the adoption of deep neural networks instead of SVM can also lead to VPMs with highly satisfactory predictive performance. Despite their promising results, these techniques are too expensive in terms of memory and execution time, due to the nature of their features (i.e., long sequences of text terms), which restricts their practicality.

The combination of software metrics with text mining has been also found promising. In [109] a sophisticated two-tier composite approach called VULPREDICTOR is proposed to predict vulnerable files. VULPREDICTOR [109] analyzes text features along with software metrics and is built on top of an ensemble of classifiers. The software attributes of the analyzed component are given as input to a group of six classifiers, the outputs of which correspond to the probability of the component to be vulnerable. These outputs are in turn processed by a meta-classifier to reach the final decision on whether the component is vulnerable or not. VULPREDICTOR outperforms the state-of-the-art approaches proposed by Walden et al. [110], which use either software metrics or text features for vulnerability prediction, but not both. The evaluation results suggest that the combination of different types of inputs and machine learning algorithms may lead to prediction models of higher accuracy.

2.2.3 Static Analysis

Limited attempts can be also found in the related literature regarding the ability of security-related static analysis alerts to predict the existence of vulnerabilities. The idea of using static analysis alerts for the identification of attack- or vulnerability-prone software components was inspired by Gegick and Williams [96]. Based on this concept Gegick et al. [111] constructed several VPMs using ASA alerts density, code churn, and lines of code as inputs, while different combinations of these features were also considered. Recursive partitioning and logistic regression was employed for the construction of these models. An empirical evaluation of the produced VPMs on a large commercial telecommunication software system revealed that the model based on ASA alerts density and code churn was the best predictor, being able to detect 100% of the attack-prone components, with 8% false positive rate (FPR). This indicates that ASA alerts can be used effectively for vulnerability prediction, while their combination with other vulnerability indicators (e.g., software metrics) may also be promising. In [66] a replication of the work presented in [111] on a different code base (i.e., a large Cisco software system) led to different observations. In fact, the FPR of the produced model was found to be higher, which suggests that the selection of the code base may influence the predictive performance of the produced models.

Moreover, several recent studies have highlighted the need for refining existing VPMs by using security-specific metrics and ASA alerts as inputs in order to improve their accuracy [112]. To this end, Yang et al. [113] proposed a novel VPM that uses security-specific warnings produced by an ASA tool to predict vulnerable components. In particular, the LDRA static analysis tool suite [114] is employed to check the conformance of the product's source code to the CERT-C secure coding standard [115, 116]. The evaluation results show that the proposed approach may lead to an improvement of up to 5% in terms of accuracy compared to the state-of-the-art models proposed in [110]. This suggests that the adoption of security-specific warnings, and especially their combination with software metrics, may be beneficial for the production of accurate VPMs.

2.2.4 Comparison of Existing Models

Finally, different empirical studies have shown that text mining-based models exhibit better predictive performance in comparison to other state-of-the-art techniques [27,67,110]. However, they perform poorly in cross-project prediction, which indicates that they are highly project-specific [110], while an excessive amount of time and memory is required for their construction and regular application [27,67]. Hence, VPMs that use software metrics (such as complexity, code churns, etc.) and density of ASA alerts may be a more viable solution in practice [67], as they are less expensive to build and apply [67], and they perform slightly better in cross-project prediction [110]. This highly suggests that the adoption of security-related ASA alerts and statically collected software metrics may be a promising approach for the construction of more accurate, as well as practical VPMs. Hence, future research attempts should focus in this direction.

2.2.5 Open Issues and Contributions

As can be seen by the previous analysis, despite the significant research attempts that have been conducted in the field of vulnerability prediction until today, there are still many open issues that require further investigation. First of all, existing empirical studies were based on a highly limited number of software applications, which hinders their reliability, as well as the generalizability of their results. In particular, the vast majority of them were based on one or two software products, with the only exception of [65] and [106] that used 14 open-source web applications and 20 Android applications respectively for the needs of their analysis. Hence, the replication of existing studies using larger code bases is considered necessary, in order to evaluate the correctness and the generalizability of their conclusions.

In addition, the majority of the existing research attempts focused on the ability of software metrics [50–59] and text features [95,105–107] to indicate the existence of security issues in software products. However, there are also other software-related factors with potential impact on software security that deserve individual merit. For instance, highly limited contributions exist regarding the ability of static analysis alerts [66,113] to indicate the presence of vulnerabilities in software, despite the fact that these research attempts demonstrated promising results. Furthermore, even in the case of software metrics, significant emphasis was given on CCC metrics,

neglecting other metrics that could potentially have an impact on software security. Therefore, current literature should be extended by examining the ability of previously uninvestigated factors to indicate security risks in software products.

Preceding research endeavors focused almost exclusively on the ability of different factors to indicate the existence of vulnerabilities in general, and not of specific vulnerability types. The only known attempt on this issue can be found in [23], in which the correlations between software metrics and specific vulnerabilities were studied, but the authors did not manage to reach useful conclusions. Hence, an interesting topic would be to investigate the ability of software-related factors to discriminate between different vulnerability types. Based on this, another interesting topic would be to investigate the existence of potential interdependencies between different vulnerability types, since current literature lacks such an attempt.

As far as existing VPMs are concerned, none of the already proposed techniques managed to achieve a satisfactory trade-off among accuracy, practicality and performance [113]. In fact, text mining-based VPMs (e.g. [106]) were found to be the most accurate, but they require significant time and memory both for their construction and for their application [67,106]. They also perform poorly in cross-project prediction, since they are based on highly project-specific features (i.e., keywords extracted directly from the source code) [67,110]. Although static analysis alerts have demonstrated promising results especially in cross-project vulnerability prediction [66,113], limited static analysis-based VPMs have been proposed. Therefore, further work is required for examining the ability of static analysis to construct VPMs that achieve a sufficient compensation among the aforementioned factors.

Another issue is that the datasets used in the literature for the derivation of VPMs are constructed based mainly on reported vulnerabilities of real products (e.g., [117,118]). However, not all of the vulnerabilities that a product contains are always reported, and therefore many components that are considered clean in the dataset may, in fact, be vulnerable. Moreover, the number of vulnerable files that a software product includes is often too small [117], leading to highly imbalanced datasets, which influence significantly the accuracy of the produced predictors [112]. Over- and under-sampling techniques are widely used in the related literature in order to address this issue (e.g., [66,106,113]). The usage of a highly balanced and sound dataset is expected to improve the accuracy of the produced VPMs. For this purpose, the vul-

nerability datasets provided by NIST¹ (e.g., Juliet suite [119,120]) or OWASP² (e.g., OWASP Benchmark [121]) could be used for the construction of such a dataset.

Finally, another topic that deserves attention is the model's transparency. All the existing VPMs are based on ML algorithms for providing their predictions. These algorithms do not provide insights to the developers, as the produced VPMs are treated in a black-box manner. Although the input vector that led to a specific decision is known, it is not clear how the final decision was reached by the model. In other words, it is not clear, which of the input features influenced the final decision the most. Hence, the feasibility of constructing a highly transparent (i.e., non-ML-based) VPM that demonstrates satisfactory predictive performance should be also investigated.

The present thesis attempts to address the majority of the aforementioned open issues. Specific emphasis is given on the ability of previously uninvestigated factors to indicate security risks in software products, as well as to discriminate between different vulnerability types. The feasibility of providing transparent vulnerability prediction based on information retrieved from static analysis is also studied.

2.3 Optimum Checkpoint Recommendation

Checkpointing is commonly utilized for adding fault tolerance to software applications both in a reactive and in a proactive way. As will be discussed in the present section, the application-level checkpoint and restart (ALCR) mechanism is the most efficient mechanism for building software applications that are fault tolerant from the beginning [33,34,122]. However, since it is based on the deliberate insertion of checkpoints into the source code, it requires significant expertise and development effort. The optimum checkpoint recommendation (OCR) corresponds to the selection of the optimum source code locations where application-level checkpoints should be inserted, as well as of the optimum checkpointing interval in case of repetitive processes (e.g., loops). This mechanism is highly useful during the SDLC as it helps developers make informed decisions regarding the optimum placement of the checkpoints, leading to more fault-tolerant and, thus, reliable software applications. In addition, since the arbitrary selection of

¹<https://www.nist.gov>

²<https://www.owasp.org>

the checkpoint locations and intervals is avoided, the potential negative impact of checkpointing on other system qualities is reduced. In the following sections, the related literature along with the fundamental background concepts of the overall field is provided. Emphasis is given on the open issues of the ALCR mechanism that the present thesis attempts to address.

2.3.1 Transaction-oriented Systems and Optimum Checkpoint Interval

Checkpoint and rollback/recovery is one of the most widely-used mechanisms for adding fault tolerance to software applications [33, 34, 122]. It was originally developed for enhancing the reliability of transaction-oriented computer systems (e.g., database or file systems), which are responsible for the sequential processing of incoming transactions [123]. If no fault tolerance mechanism is adopted by these systems, all the transactions need to be re-executed in case of a failure, leading to significant performance burden. According to the checkpoint and rollback/recovery scheme, at predetermined instants of time (i.e., intervals), a snapshot (i.e., a secure valid copy) of all the data that have been successfully processed so far is taken and stored in an area that cannot be affected by failures (e.g., a secondary file system) [123, 124]. This stored snapshot is called checkpoint [123, 124]. In case of a failure, a rollback/recovery is performed, during which the contents of the checkpoint are copied from the secondary memory to the main memory of the system [36] and all the transactions since the most recent checkpoint are re-executed [123, 124]. By using this scheme, the number of transactions that need to be re-executed is significantly reduced.

The major challenge of such systems is the selection of the *optimum checkpoint interval (OCI)*, that is, the time interval between two successive checkpoints that maximizes the system availability [36, 123, 124]. The term availability corresponds to the probability that the system is available for processing transactions [36, 37]. Hence, the vast majority of the research attempts in this field have focused chiefly on the selection of the optimum checkpoint interval (e.g., [35–37]), as well as on the impact that the checkpoint interval may have on other qualities of transaction-oriented systems (e.g., [123, 124]). The first attempt for determining the OCI was conducted by Young [35], who attempted to compute the optimum time interval between two successive checkpoints so as to achieve a satisfactory trade-off between the time required

for the establishment of a checkpoint, and the time required for the system to recover from a failure. The author proposed a simple yet practical formula for the calculation of the OCI, in which the optimum interval depends on the mean time to failures, as well as on the time required for the creation of a checkpoint.

The selection of the OCI is crucial, as the cost of checkpointing is observed to be high if the checkpoints are frequent [125]. In fact, in [123] the author investigated the impact that the selection of the OCI may have on the system performance. The results of the analysis revealed that the optimum value of the interval between two successive checkpoints that maximizes the system availability does not optimize its performance as well. Therefore, the OCI should be calculated in a way that a sufficient compensation between the contradictory factors of availability and performance is achieved. Another important parameter of transaction-oriented systems is their response time. Hence, in [124] the authors examined the impact that the selection of the checkpoint interval may have on the availability and the response time of transaction-oriented systems. The results of their analysis revealed that the average response time depends highly on the checkpoint interval and that the interval that minimizes the average response time is considerably different from the one that maximizes the system availability (i.e., the OCI). The authors also proposed a mathematical model for representing transaction-oriented systems that utilize the checkpoint and rollback/recovery mechanism. This model can be used both for the calculation of the OCI and for the estimation of its impact on important performance measures.

In [36] Gelenbe et al. proved that the checkpoint interval should be deterministic in order to maximize the system availability and that the OCI is a function of the system load. Based on these observations, the authors proposed a new formula for the calculation of the OCI that also considers the system load among other parameters. In [37] the same authors, in an attempt to further enhance the completeness of the OCI calculation, proposed a formula that takes time dependence into account. In particular, they showed that the OCI is a function of (i) the system load and (ii) the time-dependent failure rate. The results of their analysis also highlighted that the checkpoint interval should be deterministic in order to maximize the availability of the system, further supporting their previous observation [36].

Finally, Gelenbe et al. [125, 126] proposed a new fault tolerant mechanism, called failure (or virus) tests, which can be used in conjunction to the traditional checkpoint rollback/recovery

technique for further enhancing the reliability of transaction-oriented systems. According to their newly proposed approach, the data and the transaction trail of the system are periodically checked for errors and inconsistencies. If at least one error (or inconsistency) is detected, the system is forced to go through recovery as if a failure occurred. The authors also proposed a method for the calculation of the OCI, i.e., the optimum interval between two successive failure tests that maximizes the system availability.

2.3.2 Long-running Software Applications

Although the idea of the checkpoint and rollback/recovery has been initially proposed for transaction-oriented systems, it has been also found promising for enhancing the reliability of long-running software applications [31, 127]. Long-running software applications are considerably more complex compared to the transaction-oriented systems since even the most simple software programs consist of a tremendous number of execution states [28]. Therefore, periodically saving only the successfully processed data of a software application is not enough for ensuring its reliability. On the contrary, a safe copy (i.e., a checkpoint) of the overall execution state of the application should be taken and saved in a secondary file system that cannot be tampered by failures [122, 128]. This safe state can be used for recovering the execution of the program in case of a failure.

It should be noted that the majority of software failures are caused by design and implementation errors [28]. However, due to the high complexity of modern software products, it is impossible to guarantee their correctness, even with the most exhaustive validation and verification. As Dijkstra stated: *“testing can guarantee the presence of software bugs, but not their absence”* [129]. Hence, since software applications are inevitably bundled with such issues, effective fault tolerance mechanisms are required to enhance their reliability [28, 130, 131]. Several rollback/recovery-based fault tolerance mechanisms have been proposed over the years for enhancing the reliability of long-running software applications, including: (i) the Recovery Block scheme [28], (ii) the N-Version Programming [29], and (iii) the Checkpoint and Restart [31] technique.

N-Version Programming (NVP) [29, 30] and Recovery Blocks (RB) [28] are among the most well-known techniques for providing fault tolerance in software applications based on software

diversity [131,132]. In NVP [29,30], the software application is divided into modules (i.e., code blocks), and each module consists of multiple independently designed and developed versions of the same program. The N versions execute independently (in a sequential or a parallel manner) with the same input, and their outputs are combined, typically through a majority vote, in an attempt to produce the valid output of the corresponding module (i.e., code block). Similarly, in the RB scheme [28] the system is again divided into blocks of code called recovery blocks, and independent versions for each one of these blocks (which are intended to perform the same operation with the original block) are independently produced. These versions are executed sequentially, only in case their previous version fails to pass a specially designed acceptance test.

Despite the benefits of the RB and NVP approaches in enhancing the reliability of software applications, their high development costs and the significant overheads they introduce, restrict their adoption to very critical applications (e.g., safety-critical applications), in which the reliability is the most important factor. These costs were formally evaluated by Gelenbe et al. in [133], where the trade-off between the benefit of adopting this type of fault tolerance mechanisms and the loss in efficiency that they provoke for a given software application was investigated. In particular, the authors examined the impact that the adoption of the RB mechanism may have on the average execution time of a software application. Several attempts have been made in order to address the main drawbacks of these two approaches and enable their adoption in general purpose long-running software applications [130–132, 134, 135], but their associated costs are still high.

Checkpoint and Restart (CR) is a fault tolerance mechanism widely used for enhancing the reliability of long-running software applications [31,33,34], since it introduces significantly less overheads and development costs, compared to other common rollback/recovery fault tolerance mechanisms (e.g. [28,29]). A CR mechanism is responsible for keeping a safe copy of the current execution state of the software application, as well as for using this safe copy for restoring the application in case of a failure. According to [34], three types of CR exist, which are (i) the system-level CR, (ii) the library-level CR, and (iii) the application-level CR, each one of them having its own strengths and shortcomings.

Both system- and library-level CR techniques are reactive approaches for adding fault tolerance since they allow checkpointing of software applications without requiring any modification to

their source code. However, their major shortcoming is that they create checkpoints with large memory footprints, as the entire execution state of the application and the operating system processes is saved, which inevitably contains redundant information. Two representative examples of system- and library-level CR tools are BLCR [136] and DMTCP [137] respectively.

When the CR is built within the application itself, it is called application-level CR (ALCR) [32, 34]. Unlike its counterparts, it necessitates changes to the source code of the applications in order to define (i) the locations of the checkpoints, (ii) the checkpointing frequency (i.e. interval), and (iii) the data that should be checkpointed. Although it requires significant development effort, it is considered the most efficient CR approach [33, 34, 122], as it allows the creation of checkpoints with smaller memory footprints, since the minimum amount of information required for restoring the application state is essentially saved. A great number of tools for implementing ALCR in software applications can be found in the related literature [32]. A more detailed description and comparison of these types of CR can be found in [31, 32].

Several CR tools and libraries are available for ensuring the reliability of single-process software applications, including the well-known: libckpt [138], BLCR [136], and Condor [139]. However, current research in the field focuses chiefly on how to incorporate the CR mechanism into long-running multi-process HPC applications, due to the increasing reliance and usage of HPC systems, which renders the reliability of their applications important. Such applications are characterized by expensive and time-consuming computations, and therefore excessive re-computation should be avoided in case of a failure [122]. For these applications, a distributed CR scheme should be employed, in which the checkpoints of the individual processes that constitute the parallel job should be effectively combined in order to create consistent recovery states of the overall parallel application.

The most common approach for incorporating the CR mechanism into the HPC applications is by integrating it into libraries that are required for the implementation of such applications, like OpenMPI (e.g. [127]), OpenCL (e.g. [140]), OpenMP (e.g. [122, 128]) and CUDA (e.g. [141]). For instance, in [142] the authors extended the OpenMPI library, which is commonly used by HPC applications to achieve inter-node communication between processes, in order to support the CR fault tolerance mechanism. Contrary to previous fault tolerant MPI implementations that were characterized by complicated and difficult to use interfaces [127], the proposed implementation manages to automate the process of constructing the global checkpoint of the parallel

application, increasing in that way its usability. In [141] the authors proposed CheCUDA, a tool for enabling the checkpointing of CUDA applications. The proposed tool allows the effective checkpoint and restart of CUDA applications, while it also enables the migration of processes between heterogeneous nodes, enabling in that way task scheduling. However, it requires the recompilation of the CUDA applications in order to become fault tolerant. In [140], the same authors proposed CheCL, a tool for incorporating CR into OpenCL applications, since common CR libraries fail to checkpoint processes that use OpenCL. Contrary to CheCUDA [141], CheCL does not require the modification of the application source code. Instead, the proposed tool monitors the execution of the software application and all the API calls are forwarded to an API proxy, which stores all the information required for restoring OpenCL objects. The application, which is decoupled from OpenCL calls, is then checkpointed using the BLCR [136] library.

It should be also noted that, apart from HPC systems, according to a recent survey [143], among the available fault tolerance mechanisms (e.g. [28, 29, 31]), CR is commonly used for implementing fault tolerance in modern cloud-based systems, which are characterized by complex and dynamically growing architecture [144, 145]. The CR mechanism is normally utilized in such systems for restarting applications in case of failures (e.g. [146, 147]), as well as for migrating tasks and applications from one node of the cloud to another (e.g. [148]) when special circumstances impose it (e.g. node unavailability).

As already mentioned, ALCR is the most efficient CR mechanism as it incurs minimum overhead compared to its counterparts, but it requires significant development effort, which hinders its adoption in practice. To this end, Rodriguez et al. [128] proposed CPPC, a tool for providing ALCR to message passing applications. The tool manages to reduce the manual effort required by the developers, as it identifies the safe points of the applications where checkpoints should be inserted (i.e., code locations with no inconsistencies), and automatically implements the checkpoints. In fact, it identifies long-running loops and automatically inserts checkpoints at the first safe point of each loop. Losada et al. [122] proposed an application-level checkpointing solution for hybrid MPI-OpenMP applications, in which MPI is used for inter-node communication, while OpenMP for intra-node communication between processes. In fact, the proposed solution is an extension of the CPPC [128] tool, which allows checkpointing of applications implemented using either MPI or OpenMP, in order to support hybrid MPI-OpenMP applica-

tions. Since it is an ALCR approach the build-in source-to-source compiler of the CPPC [128] tool is used, in order to reduce the development effort required for the implementation of the application-level checkpoints.

Shahzad et al. [33] proposed CRAFT, a library for incorporating the application-level CR fault tolerance mechanism to software applications implemented in C++. Similarly to CPPC [128] and [122], the proposed library aims to reduce the development cost associated with the ALCR mechanism, by allowing the identification of expensive loops and the automatic insertion of the application-level checkpoints. The CRAFT library also provides mechanisms for reducing the CR overhead. These mechanisms are: (i) asynchronous checkpointing [149], and (ii) Scalable CR library [150]. In a recent research attempt, Arora et al. [34] proposed ITALC, a tool that helps the developers in semi-automatically re-engineering their applications to insert the code for the implementation of ALCR mechanism, without compromising their productivity. ITALC identifies hotspots (e.g., expensive loops) where checkpoints can be inserted and prompts the user in order to select which of those hotspots should be checkpointed. The tool can be applied to applications written in C/C++ or even parallel applications based on MPI OpenMP or even hybrid applications.

2.3.3 Open Issues and Contributions

From the previous analysis, it is clear that CR is the most widely-used mechanism for adding fault tolerance to software applications. Several libraries and tools have been proposed for incorporating the CR mechanism both to single-process (e.g. [136, 138, 139]) and multi-process (e.g. [122, 128, 140, 141]) long-running software applications. Among the CR mechanisms, ALCR is considered the most efficient, since it leaves the minimum memory footprint, but it requires significant development effort and expertise for its implementation [33, 34].

In order to address this issue, the majority of the existing ALCR libraries and tools provide recommendations regarding the locations in which application-level checkpoints should be inserted (i.e., long-running loops), while they also assist developers in automatically (or semi-automatically) inserting the checkpoints into the source code of the application. However, none of them provides recommendations regarding the inter-checkpoint interval, which is manually selected by the developers of the applications. Nevertheless, since the ALCR mechanisms

requires the insertion of a non-trivial number of library-specific instructions for the implementation of the checkpoints (see Section 5.1), not optimally selecting the checkpoint interval may introduce significant overheads in important QoS parameters of the application like execution time and energy consumption, especially in systems that failures are rare occurrences. Despite the importance of this issue, no research attempts can be found in the related literature examining the impact that the selection of the checkpointing interval of application-level checkpoints may have on important qualities of a software application, nor for facilitating the selection of the optimum checkpointing interval.

To this end, the present thesis attempts to formally assess the impact that the value of the checkpoint interval has on the performance (i.e., execution time) of software applications that adopt the ALCR fault tolerance mechanism, which is an important QoS parameter especially for time-critical applications that run frequently and repeatedly. It also attempts to provide a mathematical model for determining the optimum checkpoint interval, i.e., the interval between two successive checkpoints that optimizes the performance of software applications.

Chapter 3

Software Metrics as Indicators of Specific Vulnerability Types

Software metrics have been widely studied in the related literature regarding their ability to indicate the existence of vulnerabilities in software [50–59]. However, as stated in Section 2.2.5, there are still many open issues that require further investigation. Firstly, previous research attempts were based on a highly limited number of software products, which hinders their reliability, as well as the generalizability of their results, whereas extensive attention has been given on complexity, coupling, and cohesion (CCC) metrics, neglecting others with potential relevance to software security. More importantly, almost none of the existing research endeavors focused on the capacity of software metrics to discriminate between different types of vulnerabilities, while the existence of potential interdependencies among different vulnerability types has not been studied yet.

To this end, the main purpose of the research work presented in this chapter is (i) to investigate the discriminative power of software metrics regarding different vulnerability types, and (ii) to examine whether potential relationships exist among different types of security issues. For better understanding, these objectives can be summarized in the following two research questions:

Part of the work that is presented in this chapter has appeared in the proceedings of the 2017 International Conference on Computational Science and Computational Intelligence (CSCI 2017) [151].

RQ₁: *Can software metrics indicate the existence of specific vulnerability types?*

RQ₂: *Are there any interdependencies between different vulnerability types?*

In addition to the aforementioned objectives, the research work presented in this chapter also attempts to evaluate on a much larger code base the common belief that software metrics can be used as indicators of software vulnerabilities in general. A positive outcome of this evaluation will suggest that software metrics are sufficient indicators of software security, and, in turn, that they could potentially be used for enriching security assessment models and techniques (like the one presented in Chapter 4).

For the purposes of the present research attempt, an empirical study was conducted, based on 100 popular open-source Java applications. A wide range of software metrics (including previously uninvestigated metrics) was calculated for each project of the code base. Several security-specific and non-security-specific weakness categories were constructed by grouping the rules of a popular source code analyzer, which was also employed for quantifying the densities of the weakness categories for each project of the code base. Correlation analysis was then applied to find potential relationships between the selected software metrics and the defined weakness categories (both security- and non-security-specific), as well as between the weakness categories themselves.

The rest of the chapter is organized as follows: Section 3.1 describes the setup of the present empirical study. Section 3.2 presents the results of the experiments along with a discussion of their main observations. Finally, Section 3.3 summarizes the work conducted, and discusses the novelty of the present research attempt.

3.1 Experiment Setup

3.1.1 Code Base Construction

The first step of the empirical study was the construction of a reliable code base of software products. For this purpose, 100 popular Java applications (including popular libraries like Xerces, HyperSQL, etc.) were retrieved from the Maven Repository [152], which is the largest

online repository of Java libraries. These libraries are widely used by millions of developers around the world, and thus they constitute sufficient representatives of real-world software products, as well as sufficient sources of vulnerability information. The final repository comprises 22,259 classes (i.e., source code files), which correspond approximately to 6.5 million lines of code (LOC). This enhances the experiment reliability and renders the present empirical analysis as the largest analysis found in the relevant literature so far. It should be noted that this code base was also utilized for the calibration of the security assessment model that is presented in Chapter 4.

3.1.2 Software Metrics Selection

A wide range of software metrics was selected for the purposes of the present analysis, which were retrieved from the well-known metric suites of Chidamber and Kemerer (CK) [153] and QMOOD [78]. The selected metrics, along with a short description retrieved from [153] and [78], are presented in Table 3.1.

Table 3.1: Software Metrics

Metric	Description
<i>Complexity Metrics</i>	
WMC	<i>Weighted Methods per Class</i> : The total number of methods that a class contains weighted by their complexity values. As the tool assumes that all the methods have the same complexity (i.e., value of 1), the metric actually corresponds to the total number of methods defined in a class.
NOC	<i>Number of Children</i> : The number of immediate descendants (i.e., children) of a class.
SLOC	<i>Source Lines of Code</i> : The total number of executable lines of code of a class.
<i>Coupling Metrics</i>	
CBO	<i>Coupling Between Objects</i> : The total number of classes coupled to a given class. This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.
Ca	<i>Afferent Coupling</i> : The total number of other classes that call methods of the given class.
Ce	<i>Efferent Coupling</i> : The total number of other classes that their methods are called by the given class.

CBM	<i>Coupling Between Methods</i> : The total number of parent classes to which a given class is coupled. A class is coupled to its parent class if one of its inherited methods is functionally dependent on the new or redefined methods in the class.
IC	<i>Inheritance Coupling</i> : The total number of new or redefined methods of a class to which all its inherited methods are coupled.
<i>Cohesion Metrics</i>	
LCOM	<i>Lack of Cohesion in Methods</i> : The number of methods pairs in a class that are not interrelated through the sharing of some of the class' fields.
CAM	<i>Cohesion Among Methods</i> : This metric computes the relatedness among methods of a class based on their parameter lists. A metric value close to 1.0 is desired. (Range 0 to 1).
<i>Security-related Metrics</i>	
NPM	<i>Number of Public Methods</i> : The total number of methods in a class that are declared as public.
DAM	<i>Data Access Metric</i> : The ratio of the number of private or protected fields to the total number of fields declared in the class. A high value for DAM is preferred. (Range 0 to 1).
<i>Other Metrics</i>	
MFA	<i>Measure of Functional Abstraction</i> : This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class.

Previously uninvestigated metrics were included in our analysis. In particular, new CCC metrics like CBM and IC were used, while others with an obvious impact on software security, like DAM and NPM, were also considered. In brief, DAM is a measure of a class' encapsulation, while NPM is actually the number of a class' entry points.

For the calculation of the selected metrics the CKJM Extended [154] tool was employed, which is an open source tool able to calculate a wide range of metrics (including those defined in [153] and [78] suites), by processing the bytecode of Java files. The tool was applied to collect all the metrics described in Table 3.1 for each project of the code base. Similarly to [65], the metrics were collected at system-level of granularity, as the main purpose of the study was to investigate whether specific characteristics of software products can indicate the existence of

specific vulnerability types, and not the existence of vulnerable components. In fact, since the tool calculates the metrics at class-level of granularity, their average value was taken in order to be transferred at the system level.

At this point, it should be noted that these metrics were selected for the purposes of the present empirical study because they are standard metrics widely used in the related literature, and therefore in order for our results to be comparable to those of similar studies. However, some of these metrics may not be well defined in the literature, while their correlation to vulnerabilities may be unclear. A representative example is the SLOC metric. Although more lines of code constitute the code longer and more time-consuming to read, it does not necessarily mean that it is more complex, since the length of the code may have been increased in order to reduce its complexity. Similarly, the size of an application may not necessarily positively correlate to vulnerabilities. For example, the additional code may correspond to security mechanisms implemented for strengthening the security of the application. Therefore, through the present empirical study we expect to evaluate the findings of previous studies regarding the relation of these metrics to security issues.

3.1.3 Weakness Categories Construction

Automated static analysis was employed for the detection of the total number of potential vulnerabilities that the selected libraries contain. More specifically, the PMD ASA tool [155] was selected, which is an open-source static code analyzer that commonly used for security auditing purposes. In fact, it is included by NIST¹ in its list of static analysis tools that are recommended for detecting common vulnerabilities. In brief, PMD searches in the source code for violations of specific rules that correspond to best coding practices. If a violation is identified, an alert is produced, which is added in a broader evaluation report.

All the rules provided by PMD for Java programming language were carefully examined and grouped into several weakness categories based on their relevance. In fact, 233 rules were inspected and grouped into 13 weakness categories. In order to ensure that safe security-specific conclusions can be reached by the analysis, both security- and quality-specific weakness categories were considered.

¹<https://www.nist.gov/>

Despite the ability of PMD to detect issues with security implications, it does not provide any ready-made security-specific groups of rules (i.e., rulesets), nor information regarding the security relevance of its rules. Hence, to ensure the correct definition of the security-specific weakness categories we used information from the *Common Weakness Enumeration (CWE)* [41], which is a well-accepted dictionary of common coding and design issues that may affect software security.

In particular, a mapping between PMD rules and CWE entries was performed. In fact, for each one of the rules provided by the PMD static analysis tool, the CWE database was manually searched to find a relevant entry. For example, the PMD rule “*Avoid Catching Generic Exception*” was mapped to the CWE entry: “*CWE-396: Declaration of Catch for Generic Exception*”. This mapping led to the construction of 8 security-specific weakness categories, namely *Misused Functionality*, *Resource Handling*, *Exception Handling*, *Adjustability*, *Synchronization*, *Null Pointer*, *Logging* and *Assignment*.

Subsequently, the PMD rules that could not be mapped to security-specific CWE entries were considered non-security-specific bugs, and they were grouped into quality-specific weakness categories based on their relevance. In fact, 5 quality-specific weakness categories were defined, namely *Comprehensibility*, *Redundancy*, *Structuredness*, *Utilizability* and *Abstraction*. A description of the security- and quality-specific weakness categories that were defined through the aforementioned process is provided in Table 3.2, along with the most representative CWE category or entry that better describes each weakness group.

It should be noted that the mapping between PMD rules and CWE entries was necessary for the construction of the Security Assessment Model (SAM) that is presented in the Chapter 4 of the present report, and especially for the derivation of the model’s weights (see Section 4.1.2.2). In fact, the security-specific weakness categories were used as the static analysis-based properties (i.e., vulnerability categories) of the proposed model.

Table 3.2: Weakness Categories

Weakness Category	Description	CWE
<i>Security-specific Weakness Categories</i>		
Null Pointer	Contains rules that are able to detect issues regarding null pointer dereference. A NULL Pointer dereference typically leads to system crash.	CWE-476: NULL Pointer Dereference

Assignment	Contains rules that are able to detect variable assignment and declaration issues with security implications (e.g., local variables that are not declared final can be used as entry points by the attackers).	CWE-485: Insufficient Encapsulation
Exception Handling	Contains rules that check for improper exception handling. Improper exception handling may lead to system crash or disclosure of system information to the users [156, 157].	CWE-388: Error Handling
Resource Handling	Contains rules that check for improper management of system resources (e.g., memory, connections, etc.). Improper resource handling may lead to degradation of service or even denial of service.	CWE-399: Resource Management Errors
Logging	Contains rules that check for incorrect usage of logging functionality. Incorrect logging may lead to omission of important incidents, while misplaced logging commands (e.g., inside loops) may increase the program tardiness.	CWE-778: Insufficient Logging
Adjustability	Contains rules that check the existence of hard-coded security sensitive information. These hard-coded values may become available to attackers if the code is ever disclosed.	CWE-547: Use of Hard-coded, Security-relevant Constants
Misused Functionality	Contains rules that check for misused functions that are provided by the programming language or widely-used APIs. The security issues caused by misused APIs constitute a highly acknowledged problem in software security literature. [13]	CWE-227: Improper Fulfillment of API Contract ('API Abuse')
Synchronization	Contains rules that check for synchronization (i.e., timing) issues. Improper synchronization may lead to important problems like deadlocks and race conditions, which may cause severe security breaches, such as denial of service or unauthorized access respectively.	CWE-662: Improper Synchronization
<i>Quality-specific Weakness Categories</i>		
Comprehensibility	Contains rules that evaluate the readability and understandability of the source code. Poor comprehensibility is an indicator of poor quality (i.e. the product has not been carefully developed and maintained).	CWE-398: Indicator of Poor Code Quality
Redundancy	Contains rules that check for useless or redundant code and unnecessarily complicated expressions. High level of redundancy makes the code difficult to understand and maintain.	CWE-398: Indicator of Poor Code Quality
Utilization	Contains rules that check for the existence of unused variables and methods, or unreachable (i.e. dead) code. Violations of such rules may be indicators of deeper problems.	CWE-561: Dead Code

Abstraction	Contains rules that check for incorrect implementation of abstraction. Violations of such rules may indicate that the developers do not have a good understanding of object-oriented (OO) programming concepts, while it also hinders the component reusability.	CWE-398: Indicator of Poor Code Quality
Structuredness	Contains rules that check for violations of the structural programming principle. Violations of such rules, (e.g. multiple return statements, misplaced break statements etc.) may indicate that the developers do not have a good understanding of structural programming concepts, while the code becomes more complex.	CWE-398: Indicator of Poor Code Quality

The PMD tool, configured to identify the aforementioned types of weaknesses, was applied to the software products of the manually constructed code base. For each product, the results of the static analysis were aggregated in order to calculate the static analysis vulnerability density (SAVD) (see Section 2.1.2) of each weakness category, following an approach similar to [65] and [68]. It should be noted that the SAVD of the quality-specific weakness categories corresponds to the density of static analysis alerts.

3.1.4 Correlation Analysis

In order to provide answers to the defined research questions, correlation analysis was employed. Spearman's rank correlation coefficient (ρ) [158] was selected, as it is a non-parametric test that is not sensitive to outliers. It also does not assume any distribution for the studied data, which is important in our case as our data did not seem to follow any known distribution. To interpret the strength of the correlations Coehen's et al. [159] suggestion was used. According to Coehen et al. [159], a correlation less than 0.3 is considered weak, between 0.3 and 0.5 is considered moderate, and above 0.5 is considered strong. Finally, to ensure that the observed associations did not occur by chance, the correlations were tested for statistical significance. For this purpose, the *p-value* of each correlation was examined. A *p-value* of 0.05 means that we are 95% confident that the observed association has not occurred by chance. Hence, the statistical significance of the observed correlations was examined at 95% level of confidence. The Spearman's rank correlation coefficient has been widely used in the related literature for finding relationships between software metrics and vulnerabilities (e.g., [50, 52, 65, 73]). All the

required statistical analyses were performed using R [160].

3.2 Results and Discussion

3.2.1 Correlation between Software Metrics and Weakness Categories

In order to evaluate RQ_1 , correlation analysis was employed using Spearman's ρ (see Section 3.1.4), between the values of the selected metrics and the vulnerability densities of the identified weakness categories. The results of the analysis are presented in Table 3.3. To facilitate the readability of the correlation matrix different colors are used to denote the strength of each correlation based on the Coehen's et al. [159] suggestion (see Section 3.1.4). In particular, the values marked with black, green, and red color correspond to weak, medium and strong correlations respectively. Moreover, statistically significant correlations are marked with an asterisk. **It should be noted that we emphasize only on statistically significant positive correlations as they provide more intuitive results.** Due to the large size of the correlation matrix, these values are marked within rectangles in order to help the readers focus their attention more easily.

Table 3.3: Correlations among Software Metrics and Weakness Categories

	SLOC	CBO	WMC	NOC	NPM	DAM	Ca	Ce	LCOM	MFA	CAM	IC	CBM
<i>Quality-specific Weakness</i>													
Comprehensibility	-0.49*	-0.32*	-0.40*	-0.20	-0.20*	-0.26*	-0.36*	-0.29*	0.23	0.43 *	-0.01	-0.05	-0.50*
Redundancy	0.25	0.07	0.15	0.22 *	0.16	0.28	0.16	0.04	-0.11	-0.27*	0.16	0.18	0.16
Structuredness	0.17	-0.05	0.03	0.23 *	0.02	-0.04	0.20	-0.12	-0.05	-0.22*	-0.22*	-0.20*	0.02
Utilization	-0.11	-0.26*	-0.21*	-0.14	-0.11	-0.14	-0.17	-0.27*	0.16	0.15	0.08	0.07	-0.08
Abstraction	0.06	-0.07	-0.07	0.19	0.01	0.17	0.05	-0.09	-0.20*	-0.00	-0.17	-0.18	-0.22*
<i>Security-specific Weakness</i>													
Misused Functionality	0.10	0.02	-0.20	0.18	-0.13	0.22 *	0.09	0.01	-0.23*	0.13	-0.13	-0.09	-0.11
Resource Handling	-0.14	-0.05	0.10	0.24 *	0.20 *	0.09	-0.01	-0.03	-0.06	-0.09	-0.12	-0.11	-0.38*
Exception Handling	0.08	0.08	0.07	0.31 *	0.10	0.00	0.20	0.07	-0.07	-0.02	-0.03	0.03	0.02
Adjustability	0.29 *	0.08	0.28 *	0.15	0.19 *	0.14	0.21 *	0.05	0.00	-0.38*	-0.03	-0.03	0.55 *
Synchronization	0.32 *	0.15 *	-0.01	0.23 *	-0.04	0.21 *	0.19	0.10	-0.20*	-0.09	0.28 *	0.29 *	0.10
Null Pointer	0.26 *	-0.00	0.14	0.29 *	0.14	0.25 *	0.17	-0.06	-0.16	-0.21*	0.07	0.15	0.21 *
Logging	0.32 *	0.31 *	0.23 *	0.18	0.20 *	0.10	0.28 *	0.34 *	-0.16	-0.17	-0.04	-0.01	0.30 *
Assignment	-0.10	-0.05	0.11	0.25 *	0.18 *	0.08	-0.00	-0.03	-0.01	-0.11	-0.10	-0.08	-0.32

* $p < 0.05$, *black*: weak positive correlation, *green*: moderate positive correlation, *red*: strong positive correlation

As can be seen by Table 3.3, remarkably more (in number) statistically significant positive correlations are observed between the selected software metrics and the security-specific weakness categories, than between the metrics and the quality-specific weakness groups. In fact, the associations of the software metrics with the quality-specific weakness categories are either

positive but not statistically significant or negative. The only exceptions are observed with NOC and MFA metrics. The former demonstrates a positive and statistically significant association with the quality-related categories of *Redundancy* and *Structuredness*, while the latter with the weakness group of *Comprehensibility*. This indicates that the selected software metrics tend to be more closely related to security- than to quality-specific issues, and hence they may be able to discriminate (albeit with weak strength) between vulnerabilities and quality-related problems.

Within the group of security-specific weakness categories almost no clearly strong statistically significant correlation can be found between the metrics and any of the selected vulnerability categories (with the only exception of CBM that is strongly associated with the Adjustability weakness category). Hence, although the selected software metrics tend to be closely related to security-specific issues, it is hard to tell whether they can discriminate between different vulnerability types, as the existing correlations are weak or at best moderate.

A closer look at Table 3.3 reveals that previously uninvestigated metrics like DAM, NPM and CBM, are more closely associated with security issues than common metrics that have extensively been studied in the literature for their ability to indicate security risks in software products. This suggests that previously uninvestigated metrics may be better indicators of software security vulnerabilities, and thus they deserve individual merit and further investigation.

From the above analysis, it is clear that only a small subset of the selected metrics was found to be closely related to specific vulnerability categories, while the strengths of the observed relationships were generally weak. This indicates that software metrics may not be used as strong indicators of specific security issues. Interestingly, the majority of the studied software metrics are in general terms more closely correlated to security-specific than to non-security-specific weaknesses, which suggests that they may be able to discriminate between security-specific and quality-specific issues. Hence, this study provides further and more reliable empirical support on the common belief that software metrics are sufficient indicators of vulnerabilities in general, as it is based on a much larger code base. Another interesting observation is that previously uninvestigated metrics like CBM and especially security-related metrics, like DAM and NPM, were found to have a more obvious relation to security issues than other commonly studied metrics. Thus, the current literature in the field of vulnerability prediction should be extended by including new metrics, and preferably metrics with obvious impact on software security.

3.2.2 Correlation among Weakness Categories

In order to provide answers to **RQ₂**, correlation analysis was applied among the vulnerability densities of the identified weakness categories. The results of the analysis are presented in Table 3.4. Interestingly, a quick inspection of Table 3.4 reveals that statistically significant moderate to strong correlations exist almost exclusively between several security-specific weakness categories. Correlations between security-specific and quality-specific categories exist, but they are generally weak or not statistically significant. A closer examination of the correlation matrix is required to investigate the validity of these observations. **In fact, emphasis is given on the statistically significant moderate and strong positive associations.** These values are marked within rectangles in Table 3.4 to enhance the readability of the table.

Table 3.4: Correlations among Weakness Categories

	Comp.	Red.	Struct.	MF	RH	EH	Util.	Adjust.	Synch.	Abst.	NP	Log.	Assign.
Comprehensibility	1.00	-0.08	0.00	0.15	0.33 *	0.10	0.41 *	-0.26*	-0.05	0.23*	-0.07	-0.10	0.31
Redundancy		1.00	0.43 *	0.19	0.13	0.20	0.31 *	0.29*	0.34	0.18	0.42	0.32	0.15
Structuredness			1.00	0.08	0.24*	0.10	0.18	0.36	0.07	0.25*	0.24	0.01	0.27*
Misused Functionality				1.00	0.25*	0.27*	0.11	0.02	0.33 *	0.28*	0.36 *	0.09	0.25*
Resource Handling					1.00	0.33 *	0.17	-0.05	0.15	0.42	0.20*	0.15	0.97 *
Exception Handling						1.00	0.06	0.14	0.45 *	0.08	0.34 *	0.19*	0.33 *
Utilization							1.00	0.05	0.07	0.29*	0.29*	0.10	0.20*
Adjustability								1.00	0.10	0.13	0.42 *	0.17	0.03
Synchronization									1.00	0.09	0.34 *	0.23*	0.15
Abstraction										1.00	0.28*	0.13	0.23
Null Pointer											1.00	0.16	0.33 *
Logging												1.00	0.17*
Assignment													1.00

MF: Misused Functionality; RH: Resource Handling; EH: Exception Handling; NP: Null Pointer

* $p < 0.05$, *black*: weak positive correlation, *green*: moderate positive correlation, *red*: strong positive correlation

More specifically, *Assignment* vulnerabilities are strongly correlated to *Resource Handling* issues ($\rho = 0.97$). In addition, a moderate correlation is observed between *Assignment* and *Exception Handling* as well as *Null Pointer* vulnerability categories. *Synchronization* issues are associated with *Exception Handling*, *Misused Functionality* and *Null Pointer* weaknesses with medium strength. *Null Pointer* weaknesses apart from *Synchronization* and *Assignment*, demonstrate a relationship of medium strength with *Exception Handling*, *Misused Functionality*, *Resource Handling* and *Adjustability* weaknesses. *Exception Handling* weakness group, apart from *Synchronization*, *Null Pointer* and *Assignment* vulnerability categories, is moderately associated with *Resource Handling* issues. Finally, *Resource Handling* weaknesses are moderately correlated to *Comprehensibility*, and, as mentioned previously, to *Assignment* and *Exception Handling* issues. It should be noted that all the aforementioned correlations are positive and statistically significant at the 95% confidence level.

From the above analysis, it is clear that statistically significant correlations may exist between different vulnerability categories. *In simple words, important interdependencies may exist between different vulnerability types.* The incorporation of non-security-specific weakness categories allowed us to reach this conclusion since highly weaker relationships were observed among vulnerability categories and quality-specific weakness groups. In fact, the correlation between *Comprehensibility* and *Resource Handling* is the only non-weak statistically significant positive correlation observed between security-specific and quality-specific weakness categories.

This constitutes the most important observation of the present study, due to the fact that the prior knowledge of specific interdependencies between different security issues can be leveraged for the production of secure software, as closely related vulnerabilities can be addressed together. For instance, if a large number of *Resource Handling* issues is detected in a software product, and there is a known relationship of this type of vulnerabilities with the *Exception Handling* vulnerability category, it would be a good practice to test the software product for *Exception Handling* issues as well. Therefore, it would be beneficial for developers to analyze their software products in order to find relationships between vulnerabilities, and subsequently exploit these interdependences during the development cycle to improve the security of their software products.

3.3 Summary and Critical Analysis

The main purpose of the work presented in this chapter was to investigate the ability of software metrics to discriminate between different vulnerability types, and to examine whether potential interdependencies exist between different types of vulnerabilities. For this purpose, an empirical analysis was conducted based on a large code base containing 100 popular Java applications retrieved from the Maven Repository.

The results of the analysis revealed that software metrics can be used only as weak indicators of specific vulnerability types, while important interdependencies may exist between different vulnerability types. The latter constitutes the most important observation of the present study, since these interdependencies can be leveraged to improve the security of software products, by mitigating closely related vulnerabilities together. In addition, software metrics were found to be more closely related to security than to non-security issues, which suggest that they may

be sufficient indicators of vulnerabilities in general. Finally, previously uninvestigated metrics (e.g., DAM and NPM), were found to be closely related to security issues, which suggest that they may be good indicators of vulnerabilities and therefore they deserve further investigation. The last two observations are important for the present thesis, since they imply that software metrics can be used complementarily with static analysis alerts for enhancing the completeness of security assessment.

A note regarding the novelty of the present study is considered necessary. To the best of our knowledge, the work presented in this chapter constitutes the first study that extensively investigates the ability of software metrics to indicate the existence of specific types of security issues, and also that examines the interdependencies between different vulnerability types. Moreover, the present study extends the state of the art in the field of vulnerability prediction by evaluating the ability of software metrics to indicate the existence of security issues on a much larger code base, as well as by including previously uninvestigated metrics (e.g., DAM, NPM, etc.) in the analysis. To the best of our knowledge, this is the largest study so far in terms of code base size.

Chapter 4

Security Assessment Model

Measuring software security is very important for secure software development since it actually quantifies the success of the applied security policies (e.g., Microsoft’s SDL [17, 161]). In addition, it is commonly believed that “*you cannot control something you cannot measure*” [43]. As discussed in Section 2.1, although mature and well-accepted measurements of external security exist (e.g., *Attack Surface Metric* [44]), current literature still lacks a well-accepted approach for evaluating the internal security risk of software products¹ [23]. Different metrics and assessment methodologies able to measure a specific security aspect (e.g., [60, 61]), or the complete security level of software products (e.g., [88, 90]) exist, but they are either unreliable since they are based on arbitrary and subjective parameters, or they are not operationalized and thus they cannot be used in practice. In addition, the vast majority of the existing approaches lack thorough empirical evaluation, which also hinders their reliability [72].

Software security assessment is commonly treated as a subfield of software quality evaluation [73]. However, although a large number of quality models have been proposed for evaluating specific quality attributes, like *Maintainability* (e.g., [79]) and *Reliability* (e.g., [86]), limited contributions exist regarding *Software Security*. This can be explained by the fact that only recently *Software Security* was officially recognized as one of the main quality attributes of software products with the publication of the ISO/IEC 25010 [26] international standard. This opens a new area of research in software quality evaluation, i.e., for investigating whether well-established principles of quality assessment can be leveraged (or even extended) for constructing

¹It should be noted that the term software security is commonly used to describe the internal security of software products, while the external security is often termed as application security [1].

reliable security assessment models.

The main objective of a security assessment model is to provide a quantifiable expression of software security by combining several heterogeneous security metrics [72]. Static analysis alerts and software metrics constitute valuable sources of security information, and therefore promising candidates for the construction of security assessment models (see Section 2.2). In fact, static analysis, a testing approach that allows the identification of software bugs without requiring code execution [1], has been found effective in uncovering vulnerabilities, while recent studies have highlighted the ability of software metrics (e.g., complexity) to indicate the existence of vulnerabilities in software products, e.g., [50, 151] (see Section 2.2.1 and Chapter 3). Thus, an interesting topic is to investigate how these low-level metrics can be aggregated in a meaningful way, in order to produce a high-level score that reflects the overall security of software products.

To this end, in the present chapter, we propose a hierarchical security assessment model (SAM) that allows the evaluation of the internal security of software products written in Java, based on static analysis alerts and software metrics. The model, following the guidelines of ISO/IEC 25010 [26], decomposes the notion of security into a set of security characteristics (e.g., *Confidentiality*), which are further decomposed into a set of more tangible security properties (e.g., *Encapsulation*) that are directly quantifiable from the source code through low-level measures (i.e., static analysis alerts and software metrics). The model starts from the calculation of these low-level measures and, based on a set of *thresholds* and *weights*, systematically aggregates them, in order to produce a single security score (i.e., the *Security Index*) that reflects the security level of the analyzed software.

Specific emphasis was given on the *practicality* and *reliability* of the proposed model (see Section 2.1.4). The proposed model is highly practical, since it is fully-automated and operationalized in the form of individual tools [38], and therefore it can be used regularly during the overall development cycle. In addition, in order to enhance its reliability, its design parameters (i.e., *thresholds* and *weights*) were carefully elicited so as to avoid subjective information as much as possible. More specifically, the thresholds were calibrated based on a benchmark repository of 100 Java applications retrieved from Maven Repository [152] (the same that was used in Chapter 3). The weights of the model were elicited in a way to chiefly reflect the knowledge expressed by the Common Weakness Enumeration (CWE) [41], through a novel approach that

is based on popular decision-making techniques. The proposed model was evaluated through a set of experiments based on 150 real-world Java applications retrieved from GitHub and 1200 test cases retrieved from the OWASP Benchmark [121]. The ability of the model to indicate the existence of vulnerabilities in software components (i.e. classes) was also studied.

In brief, the major contributions of the work presented in this chapter can be summarized as follows: (1) a hierarchical security assessment model that allows the evaluation of the internal security level of software products; (2) a novel approach for the calculation of the weights of hierarchical models that reflect the well-established knowledge expressed by CWE; (3) an extensive evaluation of the ability of the proposed model to reflect the internal security of software products through a number of experiments on empirical data; and (4) an investigation of the ability of the proposed model to be used as the basis for vulnerability prediction.

The rest of the chapter is organized as follows: Section 4.1 introduces the proposed model and describes its derivation steps. Section 4.2 presents the implementation of the model and its integration in a broader CASE platform. Section 4.3 reports the results of our experimental evaluation. Finally, Section 4.4 provides a critical analysis of the work presented in this chapter.

4.1 Model Description and Construction

In the present section, the proposed security assessment model is thoroughly described. More specifically, we initially present the internal structure of the proposed SAM, along with the main steps that are executed for assessing the security level of software products. Subsequently, we describe in detail how the model was constructed providing sufficient justification for each option. This enhances the transparency of the proposed model and enables the construction of similar security models. The novel approach that was used for the calculation of a reliable set of weights that reflect the well-established information provided by CWE is also presented (see Section 4.1.2.2).

It should be noted that the model was constructed based chiefly on the features provided by the *Quality Assessment Tool Chain (QATCH)* [83], which is a recently proposed quality model derivation framework that allows the production of quality models that encapsulate novel concepts of software quality assessment. Therefore, the proposed SAM is based on state-

of-the-art software quality evaluation principles. In fact, the pivotal characteristics of QATCH that make it suitable for the construction of the proposed security assessment model are listed below:

- It allows the derivation of hierarchical quality models that their structure complies with the ISO/IEC 25010 [26] international standard.
- The produced quality models are based on static analysis alerts and software metrics, rendering the overall evaluation fully automated.
- It provides state-of-the-art techniques for threshold derivation and weights elicitation, allowing the production of more reliable models.

As will be discussed in the following, the platform was slightly extended, in order to consider security-specific concepts, and, in turn, to support the production of security assessment models. It should be noted that similarly to machine learning algorithms, the mechanisms provided by QATCH are meant only for “training” (i.e., calibrating) the proposed model. As a result, the actual reliability of the produced model depends on the specific information that is used for its construction.

4.1.1 Model Description

4.1.1.1 General Structure

The general structure of the proposed model is illustrated in Figure 4.1. As can be seen by Figure 4.1, the model has a hierarchical structure. It comprises four layers: (i) the layer of measures, (ii) the layer of properties, (iii) the layer of characteristics, and (iv) the layer of the overall security. In brief, the model starts with the calculation of a set of low-level security measures (i.e., static analysis alerts and software metrics) from the source code of the software product and uses their values along with a set of *thresholds*, in order to assign ratings (i.e., scores) to a group of higher-level properties (e.g., *Complexity*). These ratings are then aggregated using a weighted average scheme in order to calculate the ratings of a set of security characteristics (e.g. *Confidentiality*). Finally, these ratings are averaged, in order to

calculate the overall security score of the software product under analysis, which resides in the $[0,1]$ interval. In fact, the proposed security model aggregates the low-level security indicators in a sophisticated way, in order to obtain a single security score, i.e., the *Security Index*, which reflects the internal security level of the analyzed software product. The final security score provides a better understanding of the products security level to the developers and project managers, compared to the individual low-level measures.

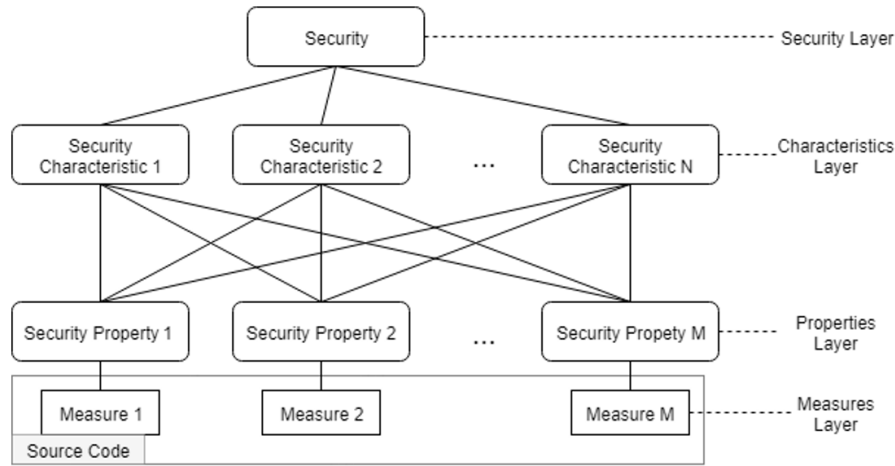


Figure 4.1: The overall structure of the proposed security assessment model (SAM)

The overall structure of the proposed model is in line with the guidelines that are provided by the ISO/IEC 25010 [26] international standard, for the construction of quality models in general. According to ISO/IEC 25010, the complex notion of software quality should be hierarchically decomposed into a set of quality characteristics (e.g., *Security*), which can be further decomposed into a set of quality sub-characteristics (e.g., *Confidentiality* and *Integrity*). Although these attributes are more tangible compared to the complete notion of quality, they cannot be measured directly from the source code of a software product, and, thus, they should be quantified indirectly through a set of low-level properties that can be directly quantified from source code.

In fact, ISO/IEC 25010 is a quality definition model [74] that acts as a blueprint for the derivation of quality assessment models. Thus, the proposed SAM is an instantiation of the ISO/IEC 25010 quality model, which allows the quantification of the quality attribute of *Software Security*. This enhances its reliability, since it is based on concepts proposed by an international standard, instead of arbitrary beliefs. Similar attempts have been conducted for other quality attributes, like *Maintainability* (e.g. [79]).

Apart from reasons of compliance with ISO/IEC 25010, the model was selected to have a hierarchical structure, since hierarchical decomposition allows the reduction of complex problems (notions) into simpler ones that are easier to understand and manage [162]. This is the main reason why hierarchical decomposition is commonly used for decision making, as well as for the construction of both quality [81] and security [90] models. In addition, the hierarchical structure allows developers and project managers to view the assessment results at different levels of abstraction, providing more fine-grained evaluation and enabling in that way root-cause analysis [79].

4.1.1.2 Specific Structure

As described in the previous section the model has a hierarchical structure. More specifically, the proposed SAM consists of three security characteristics, namely *Confidentiality*, *Integrity*, and *Availability* (see Table 4.1). The reasoning behind the selection of these characteristics is that they constitute fundamental and well-established security objectives (i.e., requirements) of software systems [163], recognized by numerous international standards as the core information security attributes (e.g., [164,165]), which together form the CIA triad of information security [163]. Another reason for their selection is that they can be assessed based on information retrieved from the application source code through static analysis. It should be noted that the security characteristic of *Non-repudiation* was also considered, but it was excluded from the final model since limited information was available for the elicitation of a reliable set of weights (see Section 4.1.2.2).

Table 4.1: The *Security Characteristics* of the proposed security assessment model. The overall concept of *Software Security* is hierarchically decomposed into three security characteristics, namely *Confidentiality*, *Integrity*, and *Availability*, which are also known as the CIA triad [163]. The definitions were retrieved from the ISO/IEC 25010 [26] international standard.

Characteristic	Description
Confidentiality	The degree to which the software product ensures that data are accessible only to those that are authorized to have access [26].
Integrity	The degree to which a software product, system, or component prevents unauthorized modification of sensitive data [26].
Availability	The degree to which a system, product, or component is operational and accessible when required for use [26].

The model also consists of 11 security properties. These properties are evaluated (i.e., quan-

tified) through code-level measures, which are either (i) security-relevant software metrics, or (ii) densities of security-specific static analysis alerts. As can be seen by Figure 4.1, each property is quantified by a single code-level measure, meaning that there is an “1-1” relationship between measures and properties. Hence, the model defines two types of properties, which are determined by the type of the measures that are used for their evaluation, i.e., (i) metrics-based properties and (ii) static analysis-based properties (which are also called vulnerability categories).

In particular, the model comprises four metrics-based properties, namely *Complexity*, *Cohesion*, *Coupling*, and *Encapsulation*, which are quantified through the software metrics of *WMC*, *LCOM*, *CBO*, and *MOA*. These metrics were retrieved from the well-known suits of Chidamber and Kemerer [153] and QMOOD [78], and calculated through the CKJM Extended [154] software metrics tool, which is integrated in the QATCH [83] platform. This tool was also used for the empirical analysis presented in Chapter 3. The aforementioned metrics-based properties and their corresponding metrics are described in Table 4.2.

The reasoning behind the selection of these metrics-based properties is that there is sufficient empirical evidence in the literature for the ability of software metrics to predict the existence of security vulnerabilities in software products (e.g., [50, 51, 54–58]). The capacity of software metrics to indicate the existence of security issues in software products was also verified by the results of the empirical study that was presented in Chapter 3. It should be noted that the empirical study presented in this chapter, was based on the same benchmark repository of software applications that is used for the calibration of the present model (see Section 4.1.2.1).

The model also contains seven static analysis-based properties, namely *Null Pointer*, *Assignment*, *Exception Handling*, *Resource Handling*, *Logging*, *Misused Functionality*, and *Synchronization*. These static analysis-based properties are, in fact, the security-specific weakness categories that were defined in Section 3.1.3 for the purposes of the empirical study (excluding the *Adjustability* category). In brief, these weakness categories were constructed by properly grouping the security-relevant rules of the PMD static code analyzer, which is also integrated in the QATCH [83] platform. Since these weaknesses correspond to potential vulnerabilities, the produced weakness groups are also termed vulnerability categories.

As mentioned in Section 3.1.3, in order to ensure the correct definition of the weakness categories, a mapping between PMD rules and CWE entries was performed. In fact, 233 PMD

Table 4.2: The software metrics-based properties of the proposed security assessment model, along with the specific metrics that are used for their quantification. These metrics are retrieved from the Chidamber and Kemerer [153] and QMOOD [78] metric suits, while they are quantified using the CKJM Extended [154] tool.

Property	Description	Metric
Complexity	The level of the logical complexity of the software product.	<i>WMC (Weighted Methods per Class)</i> : The total number of methods that a class contains weighted by their complexity values.
Cohesion	The degree to which a software product satisfies the separation of concerns principle.	<i>LCOM (Lack of Cohesion in Methods)</i> : The number of method pairs in a class that are not interrelated through the sharing of some of the class' fields.
Coupling	The level of independence between the modules of the software product.	<i>CBO (Coupling Between Objects)</i> : The total number of classes coupled to a given class.
Encapsulation	The degree to which the software product avoids user-defined information.	<i>MOA (Measure of Aggregation)</i> : This metric is a count of the number of data declarations that are user-defined.

rules for Java programming language were inspected and associated with relevant CWE entries. Representative examples of this mapping is provided in Table 4.3. Based on this mapping the PMD rules were grouped into the aforementioned vulnerability categories. A description of these categories is provided Table 3.2, along with the most representative CWE ID that better describes each category.

Table 4.3: Mapping between PMD rules and CWE entries - Representative Examples

PMD Rule	CWE Entry
AvoidCatchingGenericException	CWE-396: Declaration of Catch for Generic Exception
AvoidPrintStackTrace	CWE-209: Information Exposure Through an Error Message
CloseResource	CWE-400: Uncontrolled Resource Consumption
AvoidReassigningParameters	CWE-485: Insufficient Encapsulation
DontCallThreadRun	CWE-572: Call to Thread run() instead of start()
NullAssignment	CWE-476: NULL Pointer Dereference

It should be noted that our mapping is in line with the one provided by CodeSonar², while we also cover some additional rules. The detailed list of PMD rules that belong in each vulnerability category, along with their mapping to the CWE entries is provided on the website with the

²<https://www.grammatech.com/products/codesonar>

supporting material of the present work [38]. This mapping was also required for the derivation of the models weights through the novel approach that is described in Section 4.1.2.2.

4.1.1.3 Security Evaluation

The purpose of the security assessment model is to aggregate the code-level measures in a meaningful way, in order to calculate a high-level security score that reflects the internal security level of the software product under evaluation. This security score, which is called *Security Index (SI)*, receives a continuous value that resides in the $[0,1]$ interval. Values closer to 0 indicate bad security, while values closer to 1 indicate sufficient level of security.

Similarly to other quality models (e.g., [81, 83]), the overall assessment is performed in two sequential steps. The first step is responsible for the evaluation of the model properties, from the system-level values of their associated measures. The outcome of this evaluation is the assignment of a rating (i.e., score) to each one of the model properties. The second step is responsible for the evaluation of the models characteristics from the ratings of the properties. This step is also responsible for the calculation of the overall *Security Index* of the product from the ratings of the defined security characteristics. The security assessment procedure that is employed by the model is briefly presented in the following. For more information, we refer the reader to [83]. However, more details are provided in the following for the unique features that have been introduced to QATCH, in order to consider security-specific concepts, which are important for the derivation of security assessment models.

Initially, the model receives as input the source code of the software product that requires security assessment. Subsequently, it executes static analysis in order to detect potential security issues (i.e., static analysis alerts) that belong to the seven vulnerability categories defined in Table 3.2. It also calculates the absolute values of the four software metrics defined in Table 4.2. These measures cannot be used directly for the evaluation of the model properties since they are in a raw and low-level form. More specifically, a list of static analysis alerts is produced for each one of the seven vulnerability categories of the model, while the selected metrics are calculated at class-level (for each class of the analyzed product). Hence, appropriate aggregation and normalization should be applied, in order to bring these measures at system-level, and therefore enable the evaluation of the model properties.

In order to bring the software metrics at system-level, the aggregation and normalization scheme proposed by Wagner et al. [80] for common quality models is employed. More specifically, the system-level value of a software metric is the aggregation of its class-level values weighted by the lines of code of each class, divided by the total lines of code of the system under analysis. This normalization and aggregation approach is commonly used by other quality models as well (e.g., [24]).

As far as the vulnerability categories (i.e., static analysis-based properties) are concerned, QATCH has been extended, in order to support the calculation of the *static analysis-vulnerability density metric (SAVD)* [65,68]. As mentioned in Section 2.1.3, the *SAVD* is the total number of security-specific static analysis alerts reported by a static code analyzer for a specific software product, per thousand lines of code. For each one of the defined vulnerability categories, an individual *SAVD* is calculated, which is given by the following formula:

$$SAVD_i = 1000 \frac{N_i}{LOC} \quad (4.1)$$

where

$SAVD_i$: the *SAVD* of the i -th vulnerability category

N_i : the total number of static analysis alerts that belong to the i -th vulnerability category

LOC : the total lines of code of the software product

Therefore, although *SAVD* is normally used as an indicator of the overall security level of software products, in the present study we use *SAVD* for assessing specific vulnerability categories. This allows more fine-grained security assessment since the proneness of the analyzed software product to specific types of vulnerabilities is evaluated and considered during the overall assessment. This approach was also used in the empirical study presented in Chapter 3, as well as in recent research works [39,40]. The *SAVD* is already at system-level since the value is divided by the lines of code of the overall product.

The system-level values of these measures are used for evaluating the security properties of the model. As already mentioned, there is an “1-1” relationship between measures and properties, meaning that each property is quantified (i.e., evaluated) by a single measure (see Figure 4.1).

In particular, the system-level value of a specific measure is used for assigning a rating (i.e., score) to its corresponding property, which resides in the $[0,1]$ interval and reflects how well this security property is satisfied by the system. The concept of *utility function*, which was originally proposed in [80], is used for mapping measure values to property ratings.

Each security property has a *utility function*. A *utility function* has a partially linear form, and assigns a score between 0 and 1 to the property, based on the system-level value of its corresponding measure, and a set of measure-specific thresholds (i.e., t_l , t_m , t_u). More specifically, the *utility function* of a property P is described by the following formula:

$$f_p(m) = \begin{cases} 1 & , \quad m \leq t_l \\ \frac{0.5}{t_l - t_m}(m + t_l - 2t_m) & , \quad t_l \leq m \leq t_m \\ \frac{0.5}{t_m - t_u}(m - t_u) & , \quad t_m \leq m \leq t_u \\ 0 & , \quad m \geq t_u \end{cases} \quad (4.2)$$

In the above formula, m corresponds to the system-level (i.e., normalized) value of the code-level measure that is associated with the corresponding property P . For better understanding, the main structure of the *utility function* is presented in Figure 4.2. To enhance the subjectivity of the overall assessment, the thresholds of the *utility functions* of the proposed model are calculated based on benchmark data (see Section 4.1.2.1), instead of expert judgments.

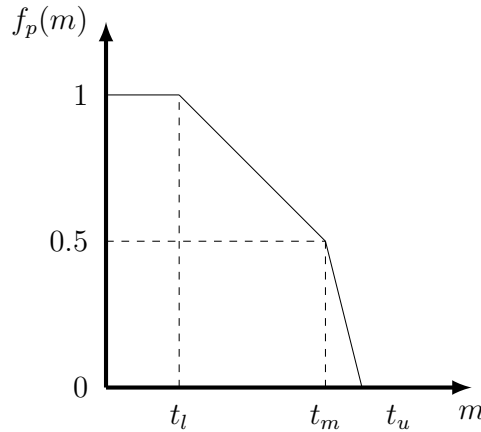


Figure 4.2: The general structure of the utility functions of a given security property P . The utility function assigns a continuous value between 0 and 1 to the corresponding property, based on the system-level (i.e., normalized) value m of its associated code-level measure and on a set of data-driven thresholds (i.e., t_l , t_m , t_u).

The next step of the overall assessment is the evaluation of the model characteristics. The

security characteristics of the model are evaluated indirectly from the ratings of the model properties, based on the impact that they have on them. As can be seen by Figure 4.1, each property has an impact on each one of the model characteristics. The ratings of the security characteristics are calculated by taking the weighted average of the ratings of the model properties. The weights constitute a quantitative expression of the impact that the model properties have on its characteristics. The weights of the proposed model were calculated based on the novel approach presented in Section 4.1.2.2, so as to chiefly reflect the knowledge expressed by CWE.

Finally, the *Security Index (SI)* of the software product under evaluation is calculated, based on the ratings of the model characteristics. The three security characteristics of the model, i.e., *Confidentiality*, *Integrity*, and *Availability*, are considered equally important for the calculation of the security level of a software product, since the primary focus of information security, in general, is the balanced protection of these three security requirements [163]. Thus, the final *Security Index* of the software product under evaluation is calculated by taking the simple average of the ratings (i.e., scores) of the three security characteristics of the model.

A discrete indicator of software security is also considered highly useful for better communicating the results of the assessment even to non-technical stakeholders, and therefore for facilitating decision-making since human brain can better perceive linguistic values compared to actual numbers. For this purpose, we propose the adoption of the classification scheme (i.e., discrete scoring scheme) presented in Table 4.4. According to this scheme, the security level of a software product can be characterized as *Very High*, *High*, *Above Normal*, *Below Normal*, *Low*, or *Very Low*, based on the actual value of its *Security Index*. These categories were defined based on the normal distribution of the *Security Indexes* of the software products that were used for the calibration of the proposed model (i.e., the 100 Java applications), an approach that is commonly used for determining discrete quality scores (e.g., SIG Model [79]). In Table 4.4, we also report the percentage of the software products of the code base that reside in each rating group. From Table 4.4, one may easily realize that the *SI* value of 0.5, equally divides the code base. In fact, about 50% of the benchmark products received an *SI* above 0.5, and obviously about 50% received an *SI* below 0.5.

Table 4.4: The recommended discrete scoring scheme of the proposed security assessment model. The last column displays the percentage of the applications of the benchmark repository that was used for the calibration of the model that belong in each Security Class.

Security Class	Security Index (SI)	Percentage
Very High	(0.8, 1.0]	10%
High	(0.6, 0.8]	17%
Above Normal	(0.5, 0.6]	20%
Below Normal	(0.4, 0.5]	26%
Low	(0.2, 0.4]	18%
Very Low	[0, 0.2]	9%

4.1.2 Model Construction

From the previous analysis, it is clear that the proposed security assessment model is based on a solid basis (i.e., well-established concepts), with respect to its structure and the performed security assessment. More specifically: (i) its structure complies with the one proposed by the ISO/IEC 25010 [26] international standard, (ii) it encapsulates well-established concepts of quality models (e.g., aggregation [79], utility functions [80], etc.), and (iii) it is based on low-level measures with known relevance to software security (i.e., static analysis alerts and software metrics). However, this is not enough for producing a reliable SAM, which provides sufficiently objective security evaluations. The reliability of the final model is determined chiefly by its design parameters, since an arbitrarily selected parameter may affect the correctness (i.e., objectivity) of the performed assessment.

From the description of the security assessment model provided in Section 4.1.1, it is obvious that two are the main design parameters of the model that need to be determined, namely (i) its *thresholds*, and (ii) its *weights*. These parameters constitute the main sources of potential subjectivity, since they are commonly determined based on expert judgments, and therefore their values should be carefully selected.

In the present section, we present in detail how the proposed security assessment model was constructed, and particularly how the model's *thresholds* and *weights* were derived. More specifically, we describe how we derived the model *thresholds* exclusively from data based on a well-established approach. We also describe a novel approach that allows the derivation of the

model *weights* based on the well-accepted information provided by the CWE knowledge base, instead of individual expert judgments.

4.1.2.1 Threshold Derivation

The thresholds of the proposed SAM are used by the *utility functions* in order to assign ratings (i.e., scores) to the model properties, and therefore they are very important for the overall assessment. These thresholds can be derived based either on expert judgments or on empirical data. The latter approach is highly preferred in the literature since it leads to the derivation of a reliable set of thresholds, which are completely free from subjective information. Among the existing threshold derivation approaches, benchmarking is widely used in the related literature for threshold derivation (e.g., [79, 80, 83]), as it allows the calculation of thresholds based exclusively on data, without requiring expert judgments, which are highly subjective. Therefore, the benchmarking approach was applied for the derivation of the thresholds of the proposed security assessment model.

For the conduction of benchmarking, a large code base of software products is required, which constitutes a representative population of widely-used and mature software products. For this purpose, the benchmark repository that was constructed for the purposes of the empirical study presented in Chapter 3 was used. In brief, this benchmark repository contains 100 Java applications that were retrieved from the Maven Repository [152] based on their popularity. These applications are highly mature and widely used by millions of developers around the world, and thus they constitute sufficient representatives of real-world software products. As mentioned in Section 3.1.1 the final repository comprises 22,259 classes (i.e., source code files), which correspond approximately to 6.5 million lines of code (LOC).

Initially, QATCH was employed in order to analyze the source code of each one of the software products of the code base, in order to calculate the system-level values of the measures that correspond to the 11 properties of the model (see Table 3.2 and Table 4.2). In particular, the normalized values of the metrics-based properties and the *SAVD* metrics of the vulnerability categories of the model were calculated, for each one of the software products found in the code base. A screenshot of the calculated measures of 20 software applications selected from the broader repository is illustrated in Figure 4.3.

Resource Handling	Exception Handling	Misused Functionality	Synchronization	Null Pointer	Logging	Assignment	Complexity	Cohesion	Coupling	Encapsulation
92.255811	7.647838131	14.99458027	2.107671926	0.963507166	0.120438396	101.951102	14.49644707	182.3562	10.00572082	0.468987113
48.991114	2.275055387	11.41101079	0.440717536	0.940991495	0.131024132	58.8179241	22.66142173	312.69656	18.24527718	4.063665817
48.653636	1.962922574	12.90160221	0.654307525	1.375723513	0.872410033	64.17246875	22.74346112	205.75853	17.91759081	4.84939183
75.859179	1.485940715	3.371942391	0.647717747	0.552465138	0.09525261	94.64299322	66.04511164	3938.1956	7.864341233	1.145584089
65.492394	4.163330665	8.967173739	0.320256205	2.56204964	2.081665332	72.37790232	29.12313851	436.62626	5.269495596	0.255244195
78.777001	1.265944491	10.36792513	0.945452215	0.320492276	1.826805974	94.27280303	22.3037786	392.86203	20.12547273	0.961540927
47.960346	5.425681559	8.372965369	0.602853507	0.870788398	1.473641905	64.37135776	30.24917945	616.52964	23.97046018	0.866233505
45.831257	0.895213836	6.238809827	0.664488621	0.812152759	0.323015302	53.98970043	24.50462373	342.06464	16.48607343	3.978044188
4.0215877	1.14902507	12.29108635	1.497214485	2.802924791	0.974930362	11.50766017	16.52416435	198.39112	20.03445334	2.219428969
4.8789572	0.856610801	14.26443203	1.340782123	1.601489758	0.037243948	10.01862197	13.47675978	52.165438	12.45750466	2.003426443
83.458447	2.384527069	11.21610881	0.441579087	2.472842886	0.353263269	89.55223881	33.07180076	824.23969	12.45164709	2.657599576
85.229647	2.163313803	10.42323923	0.114721187	2.417339288	0.032777482	95.25136231	38.30301963	1027.9391	33.86964395	3.415257918
52.835656	0.883124051	9.562577618	0.137988133	0.662343038	0.896922865	64.90961777	40.82988823	2179.1034	50.28077825	2.826535118
78.104111	0.914732828	7.036406367	0.281456255	0.7740047	0.30960188	90.89629744	36.07295346	1021.9177	18.26883294	4.171491296
96.996476	2.524854032	10.99363526	0.762716322	2.60375572	0.867918573	106.5961812	36.6170375	851.54166	13.78891168	1.646073326
75.608936	7.762598943	9.687967972	1.375263592	1.222456526	1.405825005	79.73472693	25.12826625	676.15867	14.76993368	0.526634272
79.668231	1.964422132	13.31441668	4.147113391	5.347593583	2.510094947	105.6422569	14.31845465	71.602041	4.916784896	1.107606679
74.68036	1.033876693	7.34052452	0.861563911	1.826515491	1.654202709	88.70662026	28.7546266	689.74591	22.57338801	1.914739635
56.666292	2.280560016	9.265414234	1.799903869	1.544235706	5.430391786	65.67603776	20.42558522	171.55794	20.16382193	1.887659409
64.715234	1.106642975	10.8014589	0.452009103	3.366688488	0.077932604	71.68240905	25.87614951	353.27161	6.830621279	0.136428816

Figure 4.3: The system-level values of the security properties of 20 software applications retrieved from the complete benchmark repository used for the calibration of the proposed security assessment model.

Subsequently, for each property three thresholds were calculated based on the distributions of the system-level values of their associated measures observed between the different products. More specifically, the formulas proposed by [80] were used for deriving these thresholds:

$$\begin{aligned}
t_l &= \min(\{m : m \geq Q_{25\%}(m_1, \dots, m_n) - 1.5 \cdot IRQ(m_1, \dots, m_n)\}) \\
t_m &= \text{median}(m_1, \dots, m_n) \\
t_u &= \max(\{m : m \leq Q_{75\%}(m_1, \dots, m_n) + 1.5 \cdot IRQ(m_1, \dots, m_n)\})
\end{aligned} \tag{4.3}$$

where:

m_i denotes the normalized value of the measure m of the i -th benchmark product

Q_p denotes the p -percentile

$IRQ(m_1, \dots, m_n)$ denotes the inter-quartile-range

In simple words, after removing the outliers, the minimum, median, and maximum observations of each measure were selected as the lower (t_l), middle (t_m), and upper (t_u) thresholds of the associated property's utility function respectively. The final thresholds of the proposed model that were derived through the aforementioned approach are presented in Table 4.5. For reasons of reproducibility, the full list of the calculated measures, along with the script that calculates

the thresholds based on equations (4.3) are available on the website with the supporting material of the present study [38].

Table 4.5: The final thresholds of the proposed security assessment model.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}
t_l	2.4	0	0	0	0	0	13.6	2	1	0	0
t_m	58.2	2	9.5	0.6	1.4	0.4	69.2	24.7	366.5	12.9	1.7
t_u	102	6.3	19.5	2.7	3.8	2.5	133.8	62.39	1571.2	42.3	5.5

P_1 : Resource Handling; P_2 : Exception Handling; P_3 : Misused Functionality; P_4 : Synchronization; P_5 : Null Pointer; P_6 : Logging; P_7 : Assignment; P_8 : Complexity; P_9 : Cohesion; P_{10} : Coupling; P_{11} : Encapsulation

4.1.2.2 Weights Elicitation

The model weights constitute quantifiable expressions of the impacts that the properties of the model have on its characteristics. Contrary to threshold derivation, weights elicitation cannot be data driven. Weights are commonly derived based on expert judgments, which inevitably contain subjectivity. However, being based on the opinions of individual experts is known to affect the reliability (i.e., trustworthiness) of the produced models, due to the fact that the performed assessment may not be sufficiently objective, while the expertise of the selected experts may be questionable.

Hence, contrary to previous attempts, in the present work we propose a novel approach that allows the derivation of weights that reflect the well-established knowledge expressed by CWE, instead of expert judgments. As will be discussed in the following, the proposed approach is based on well-known decision-making techniques, particularly the Analytic Hierarchy Process (AHP) [162] and the SMARTS/SMARTER [166] approach, for deriving a reliable set of weights that reflect the information provided by CWE.

The Analytic Hierarchy Process (AHP) [162] is a decision-making technique that reduces complex decisions to pair-wise comparisons. It is commonly used for facilitating the selection of the best option among a set of alternatives, based on a set of criteria that are evaluated using expert judgments. In brief, it is commonly used for providing solutions to hierarchical multi criteria decision-making problems.

AHP constitutes a suitable technique for calculating the impacts (i.e., weights) of the model

properties on the model characteristics. Suppose that the proposed model comprises N characteristics (i.e., $C_1, C_2, \dots, C_N$) and M properties (i.e., $P_1, P_2, \dots, P_M$). According to AHP, for each one of the model characteristics, a pair-wise comparison (pwc) matrix is required for the calculation of its weights. The general structure of the pwc matrix that is required for the calculation of the weights of an arbitrary characteristic C_k of the model is presented in Table 4.6.

Table 4.6: The general structure of the pairwise comparison (pwc) matrix that is required for calculating the weights of a characteristic C_k of a security model with M properties, namely $P_1, P_2, \dots, P_M$. The value of e_{ij} denotes how stronger (or weaker) the impact of P_i on the C_k is, compared to the corresponding impact of P_j . The pairwise comparison matrix is normally completed with expert judgments.

C_k	P_1	P_2	$\dots$	P_j	$\dots$	P_M
P_1	e_{11}	e_{12}	$\dots$	e_{1j}	$\dots$	e_{1M}
P_2	e_{21}	e_{22}	$\dots$	e_{2j}	$\dots$	e_{2M}
$\vdots$	$\vdots$	$\vdots$		$\vdots$		$\vdots$
P_i	e_{i1}	e_{i2}	$\dots$	e_{ij}		e_{iM}
$\vdots$	$\vdots$	$\vdots$		$\vdots$		$\vdots$
P_M	e_{M1}	e_{M2}	$\dots$	e_{Mj}	$\dots$	e_{MM}

As can be seen by Table 4.6, the rows and the columns of the pwc matrix correspond to the properties of the model. The cells of the pwc matrix should be completed with values that denote the relative impact of the pairs of properties that correspond to each cell on the selected characteristic (i.e., C_k). More specifically, the value of e_{ij} denotes how stronger (or weaker) the impact of the property P_i on the characteristic C_k is, compared to the impact of the property P_j . These values are normally completed by experts according to their opinions.

Subsequently, the normalized principal eigenvector (i.e., the eigenvector that corresponds to the maximum eigenvalue) of the final matrix is selected to be the weights vector $\bar{w}$ of the corresponding characteristic C_k . More specifically, if A is the aforementioned pwc matrix, then it can be formally written as follows:

$$A = (e_{ij}), \text{ where } : (i, j = 1, 2, \dots, M) \quad (4.4)$$

The eigenvalues and eigenvectors of the matrix are calculated by solving the equation:

$$(A - \lambda I) = 0 \quad (4.5)$$

The vector $\bar{w}$ containing the desired weights is derived by the following formula:

$$\bar{w} = \frac{\bar{w}_{\max}}{\sum_{i=1}^M w_{\max,i}} \quad (4.6)$$

where:

$\bar{w}_{\max}$: the principal eigenvector of the pairwise comparison matrix

$w_{\max,i}$: the i -th entry of the principal eigenvector

M : the length of the principal eigenvector

The main advantage of AHP is that the experts have to focus only on a single pair of properties at each time, while the interdependencies between the properties are automatically considered during the weights calculation. This approach guarantees that the produced weights will reflect the expert judgments (i.e., the information expressed formally in the pwc matrices). However, due to the fact that pwc matrices are completed based on expert judgments, subjectivity still exists. To avoid subjectivity, the weights should reflect commonly accepted impacts expressed by a wider community of experts. This can be achieved by completing the pwc matrices of the AHP approach with impact values that were derived by well-accepted sources of information, instead of individual expert judgments.

As far as the static analysis-based properties are concerned, the CWE knowledge base is used as the source of information. In particular, as described in Section 4.1.1.2, the rules of each vulnerability category of the model were mapped to relevant CWE entries. Each CWE entry contains important security-relevant information for its corresponding weakness, including its impact on well-known security requirements, such as *Confidentiality*, *Integrity*, and *Availability*. For instance, as can be seen by Figure 4.4, the *CWE-248: Uncaught Exception*, according to CWE, influences both the *Availability* and the *Confidentiality* of a given software product. Therefore, CWE was used as a knowledge base for determining the impacts of the static analysis-based properties on the security characteristics of the model.

CWE-248: Uncaught Exception

Weakness ID: 248
 Abstraction: Base
 Structure: Simple

Status: Draft

Presentation Filter: Basic

Description
 An exception is thrown from a function, but it is not caught.

Extended Description
 When an exception is not caught, it may cause the program to crash or expose sensitive information.

Relationships
 The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOf and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that the user may want to explore.

- Relevant to the view "Research Concepts" (CWE-1000)
- Relevant to the view "Development Concepts" (CWE-699)

Common Consequences
 The table below specifies different individual consequences associated with the weakness. The Scope identifies the application security area that is violated, while the Impact describes the negative technical impact that arises if an adversary succeeds in exploiting this weakness. The Likelihood provides information about how likely the specific consequence is expected to be seen relative to the other consequences in the list. For example, there may be high likelihood that a weakness will be exploited to achieve a certain impact, but a low likelihood that it will be exploited to achieve a different impact.

Scope	Impact	Likelihood
Availability	Technical Impact: DoS: Crash, Exit, or Restart; Read Application Data	
Confidentiality	An uncaught exception could cause the system to be placed in a state that could lead to a crash, exposure of sensitive information or other unintended behaviors.	

Figure 4.4: CWE provides information about the impact that each weakness has on the main security requirements of software products.

Initially, for each one of the static analysis-based properties (i.e., vulnerability categories) of the model we listed their relevant CWE entries based on the mapping between their relevant PMD rules and the CWE entries. As already mentioned, each CWE entry potentially has an impact on each one of the models characteristics. If E is an arbitrary CWE entry, we can formally define whether it has a reported impact on a specific characteristic C of the model by using the following indicator variable:

$$I_{E,C} = \text{impact}(E, C) = \begin{cases} 1, & \text{if } E \text{ affects } C \\ 0, & \text{otherwise} \end{cases} \quad (4.7)$$

Based on the identified binary impacts of the individual CWE entries, the impacts of the static analysis-based properties on the model characteristics can be quantified as follows. Suppose that a static-analysis-based property (i.e., vulnerability category) P_i contains L CWE entries, namely $E_{i,1}, E_{i,2}, \dots, E_{i,L}$. As already mentioned, each one of these entries may have a reported impact on each one of the N characteristics of the model, i.e., $C_1, C_2, \dots, C_N$. The impact (i.e., impact score) that the selected property P_i has on a specific characteristic C_k can be calculated based on the reported impacts of its relevant CWE entries (i.e., CWE-reported impacts) as

follows:

$$I_{P_i, C_k} = impact(P_i, C_k) = \frac{N_{P_i, C_k}}{\sum_{k=1}^N N_{P_i, C_k}} = \frac{\sum_{j=1}^L I_{E_{i,j}, C_k}}{\sum_{k=1}^N \sum_{j=1}^L I_{E_{i,j}, C_k}} = \frac{\sum_{j=1}^L impact(E_{i,j}, C_k)}{\sum_{k=1}^N \sum_{j=1}^L impact(E_{i,j}, C_k)} \quad (4.8)$$

where:

N_{P_i, C_k} : the number of the CWE entries of P_i (i.e., $E_{i,1}, E_{i,2}, \dots, E_{i,L}$) that have a reported impact on C_k .

$I_{E_{i,j}, C_k}$: the impact that the j -th CWE Entry of the i -th property P_i has on the k -th characteristic C_k of the model (as reported by CWE). It is given by equation (4.7).

M : the number of properties of the model.

In simple words, the impact that a property P_i has on a specific characteristic C_k corresponds to the total number of the CWE-reported impacts that P_i has on C_k , divided by the total number of the CWE-reported impacts that P_i has on any of the characteristics considered by the analysis. It should be noted that the value of $I_{E_{i,j}, C_k}$ is given by (4.7).

Based on the aforementioned approach we calculated the impact scores of all the static analysis-based properties of the proposed model (see Table 4.7), on all the characteristics of the model, namely *Confidentiality*, *Integrity*, *Availability*. It should be noted that the characteristic of *Non-repudiation*, which refers to the ability of the system to prove that specific actions have taken place and also to map these actions to their responsible entities so that they cannot be repudiated later [26], was also considered in the initial analysis, since relevant information was provided by CWE. The calculated impact scores are presented in Table 4.7. These impact scores denote the impact that the corresponding vulnerability category (i.e., property) has on each one of the identified security characteristics, according to the CWE knowledge base.

Table 4.7: The impact scores of the model properties on security characteristics (calculated based on CWE).

Property Name	Confidentiality	Integrity	Availability	Non-repudiation
Null Pointer	0.167	0.167	0.67	0
Exception Handling	0.333	0.333	0.167	0.167
Logging	0.142	0	0.142	0.714
Resource Handling	0.375	0.250	0.375	0
Assignment	0.360	0.5	0.136	0
Misused Functionality	0.25	0.625	0.125	0
Synchronization	0.23	0.462	0.308	0

The impact scores presented in Table 4.7 seem to reflect the common knowledge regarding software security, as expressed by the CWE knowledge base. For instance, *Null Pointer* issues influence more the *Availability* of a system since a null pointer dereference may lead to a system crash. Similarly, *Logging* issues influence more the *Non-repudiation* requirement, as insufficient logging may lead to omission of specific events that could be used to identify the sources of security issues. However, the absolute values of these impact scores may not be accurate enough due to the fact that they were determined based on the manual mapping of CWE entries to weakness categories. On the contrary, the relative (i.e., quantitative) relationships between these impacts seem highly intuitive and representative of the real situation, as revealed by the previous examples. Hence, to enhance the reliability of the produced model, the weights elicitation was based on the relative impacts.

In order to determine the relative impacts, the SMARTS/SMARTER [166] approach was adopted. SMARTS/SMARTER is a decision-making technique that assigns relative scores (i.e., weights) to a group of attributes based on their ranking. Each one of the assigned scores reflects the relative importance of the associated attribute compared to the other attributes. The assigned weights are fixed and depend only on the total number of the ranked attributes.

To state it more formally, suppose that we are interested in calculating the relative impact of property P_i on the characteristic C_k . Initially, all the M properties are ranked in a descending order based on their previously defined *impact scores* on the C_j characteristic (see Table 4.7). A fixed relative impact score r_{ik} is assigned to property P_i , based on the following equation:

$$r_{ik} = f_{SMARTS/SMARTER}(M, rank_{ik}) \quad (4.9)$$

where:

$rank_{ik}$: the position of property P_i in the properties ranking of C_k

M : the total number of the model's properties.

The SMARTS/SMARTER approach assigns a fixed set of weights (i.e., relative scores) to a ranked list of attributes, based on the total number of attributes (i.e., properties) and their position in the ranked list. In Table 4.8, the weights that are assigned by the SMARTS/SMARTER approach for a varying number of properties are shown.

Table 4.8: The weights assigned by the SMARTS/SMARTER approach for a various number of elements (i.e. M). The table is adapted from [166].

Rank	$M = 2$	$M = 3$	$M = 4$	$M = 5$	$M = 6$	$M = 7$	$M = 8$	$M = 9$	$M = 10$
1	0.7500	0.6111	0.5208	0.4567	0.4083	0.3704	0.3397	0.3143	0.2929
2	0.2500	0.2778	0.2708	0.2567	0.2417	0.2276	0.2147	0.2032	0.1929
3		0.1111	0.1458	0.1567	0.1583	0.1561	0.1522	0.1477	0.1429
4			0.0625	0.0900	0.1028	0.1085	0.1106	0.1106	0.1096
5				0.4000	0.0611	0.0728	0.0793	0.0828	0.0846
6					0.0278	0.0442	0.0543	0.0606	0.0646
7						0.0204	0.0335	0.0421	0.0479
8							0.0156	0.0262	0.0336
9								0.0123	0.0211
10									0.0100

To this end, each column of Table 4.7 was retrieved and its properties were ranked based on their actual impact scores. Subsequently, the SMARTS/SMARTER approach was applied and a set of fixed weights were assigned to those ranked properties. These fixed weights correspond to the relative impact scores of the properties regarding the selected characteristic (i.e., column). Table 4.9 summarizes the rankings of the properties along with the relative impacts for each one of the selected security characteristics. It should be noted that the *Non-repudiation* security characteristic was excluded from the rest of the analysis, and therefore eliminated from the final model since its impact scores did not allow us to produce a reliable ranking of the model properties. From Table 4.9, it is obvious that the assigned relative impact scores are the weights of Table 4.8 for $M = 7$.

Table 4.9: The rankings of the model’s properties for each characteristic, along with their relative impact scores, retrieved from the SMARTS/SMARTER [166] approach.

Confidentiality	Integrity	Availability	Impacts
Resource Handling	Misused Functionality	Null Pointer	0.3704
Assignment	Assignment	Resource Handling	0.2276
Exception Handling	Synchronization	Synchronization	0.1561
Misused Functionality	Exception Handling	Exception Handling	0.1085
Synchronization	Resource Handling	Logging	0.0728
Null Pointer	Null Pointer	Assignment	0.0442
Logging	Logging	Misused Functionality	0.0204

These relative impact scores were subsequently used to complete the cells of the pwc matrices that correspond to the relationships between vulnerability categories (i.e., static analysis-based properties). A simple procedure for completing these cells of the pwc matrices based on the relative scores of Table 4.9 was followed. Suppose the pwc matrix of the characteristic C_k presented in Table 4.6, and e_{ij} is the expert judgment that corresponds to the pair of properties (P_i, P_j) , where P_i is the property of the row and P_j is the property of the column. Suppose also that r_{ik} and r_{jk} correspond to the relative impact scores of the properties P_i and P_j for the corresponding characteristic C_k as retrieved by SMARTS/SMARTER (see Table 4.9). The expert judgment e_{ij} is calculated as follows:

$$e_{ij} = \begin{cases} \left\lceil \frac{\left(\frac{r_{ik}}{r_{jk}}\right)}{2} \right\rceil & \text{if } r_{ik} > r_{jk} \\ 1 & \text{if } r_{ik} = r_{jk} \\ \left\lceil \frac{\left(\frac{r_{jk}}{r_{ik}}\right)}{2} \right\rceil & \text{if } r_{ik} < r_{jk} \end{cases} \quad (4.10)$$

The above formula ensures that the final judgments will lie in the interval $[0.1, 9]$, as suggested by Saaty [162].

The aforementioned approach allowed us to complete the cells of pwc matrices that are relevant to vulnerability categories of the model based exclusively on information provided by CWE, without considering any expert judgments. Unfortunately, no similar well-accepted and reliable source of information exists regarding the software metrics-based properties. Therefore, these cells were completed by the authors of the paper based on their expertise.

Following this approach three individual pwc matrices were constructed, one for each one of the three characteristics of the proposed model, namely *Confidentiality*, *Integrity*, and *Availability*. Subsequently, AHP was applied using the produced pwc matrices and the models weights were calculated based on equations (4.5) and (4.6). As an example, the pwc matrix that was used for the calculation of the weights of *Confidentiality* is illustrated in Figure 4.5. The other two pwc-matrices are available on the website with the supporting material of the present work [38]. The final weights of the models characteristics are presented in Table 4.10. The final weights, seem to sufficiently reflect both the knowledge retrieved by CWE knowledge base (i.e., the rankings presented in Table 4.9), and the judgments provided by the experts.

	Resource Handling	Assignment	Exception Handling	Missused API	Synchronization	Null Pointer	Logging	Complexity	Cohesion	Coupling	Encapsulation	
Resource Handling	-		1	1	2	3	4	9	9	6	6	5
Assignment	-	-		1	1	2	3	6	9	6	6	4
Exception Handling	-	-	-		1	1	2	4	8	5	5	4
Missused API	-	-	-	-		1	1	3	7	5	5	3
Synchronization	-	-	-	-	-		1	2	7	4	4	3
Null Pointer	-	-	-	-	-	-		1	6	4	3	3
Logging	-	-	-	-	-	-	-		5	3	3	2
Complexity	-	-	-	-	-	-	-	-		4	1	0,5
Cohesion	-	-	-	-	-	-	-	-	-		0,25	0,125
Coupling	-	-	-	-	-	-	-	-	-	-		0,5
Encapsulation	-	-	-	-	-	-	-	-	-	-	-	

Figure 4.5: The pairwise comparison matrix used for the calculation of the model’s weights that correspond to the characteristic of Confidentiality.

Table 4.10: The final weights of the model’s characteristics as derived from the AHP [162] approach. The weights reflect both the impacts retrieved from the CWE knowledge base, and the expert judgments.

Property Name	Confidentiality	Integrity	Availability
Resource Handling	0.2164	0.0973	0.1810
Assignment	0.1739	0.1680	0.0751
Exception Handling	0.1440	0.1153	0.1164
Misused Functionality	0.1180	0.2244	0.0593
Synchronization	0.0992	0.1522	0.1393
Null Pointer	0.0805	0.0798	0.2217
Logging	0.0546	0.0517	0.0877
Complexity	0.0220	0.0321	0.0328
Coupling	0.0182	0.0147	0.0230
Cohesion	0.0279	0.0185	0.0516
Encapsulation	0.0452	0.0460	0.0122

Although the weights of the metric-based properties are characterized by subjectivity (i.e., expert judgments), we decided to include these properties in the final model for a number of

reasons. Firstly, since software metrics are indirect indicators of software security (i.e., vulnerabilities) [50, 51, 54–58, 151] (see Chapter 3), as they highlight questionable constructs from a security viewpoint, enriching the security assessment with information retrieved from software metrics is considered valuable for enhancing the completeness of the model. Secondly, due to the fact that static analysis alerts are direct indicators of vulnerabilities, higher weights were assigned to the static analysis-based properties. In particular, as can be seen by Table 4.10, the total impact of the metric-based properties on the characteristics of *Confidentiality*, *Integrity*, and *Availability*, is 11.33%, 11.13%, and 11.96% respectively. This is reasonable since software metrics were found to be only weak indicators of vulnerabilities. Hence, by keeping their impact low, metric-based properties are expected to enrich the produced *Security Index* with additional security-relevant information without predominantly determining its value. Finally, an experiment conducted on a large volume of empirical data revealed that the inclusion of metrics-based properties does not influence the overall assessment (see Section 4.3.3). In brief, the assessment results of the proposed model were found to be in high accordance with those produced by an equivalent model that omits software metrics.

To sum up, the weights elicitation approach presented in this section can be used in practice for the derivation of a reliable set of weights, which are completely free from subjective information incurred by expert judgments. This is feasible since most of the available static code analyzers, especially the commercial ones, provide a build-in mapping between the weaknesses that they identify and the relevant CWE entries, reducing in that way the required manual effort. Finally, it should be noted that the proposed approach is not limited to security models. It can be used for deriving a reliable set of weights for any hierarchical quality model, provided that a well-structured source of information (similar to CWE) is available. The subjectivity of the models weights is a long-standing issue in the broader field of software quality assessment, which hinders the reliability of the produced quality models [79, 80], and therefore limits their adoption in practice.

4.2 Implementation

To the best of our knowledge, no operationalized security assessment model able to evaluate the internal security level of software products is available in the related literature. To fill this

void, the proposed model was implemented as a standalone open-source command-line tool, which can be used in practice for security evaluation purposes. This tool is available on the web page with the supporting material of the present work [38], along with detailed instructions for its utilization. The command line version of the model allows its integration with existing continuous integration and assessment platforms, while it also enables the fully-automated and headless execution of the assessment, allowing its regular application during the development cycle. In addition, in order to facilitate the manual (on-demand) execution of the model by the developers, an offline application that offers a simple graphic user interface (GUI) is also provided. The main screen of this simple GUI is illustrated in Figure 4.6.

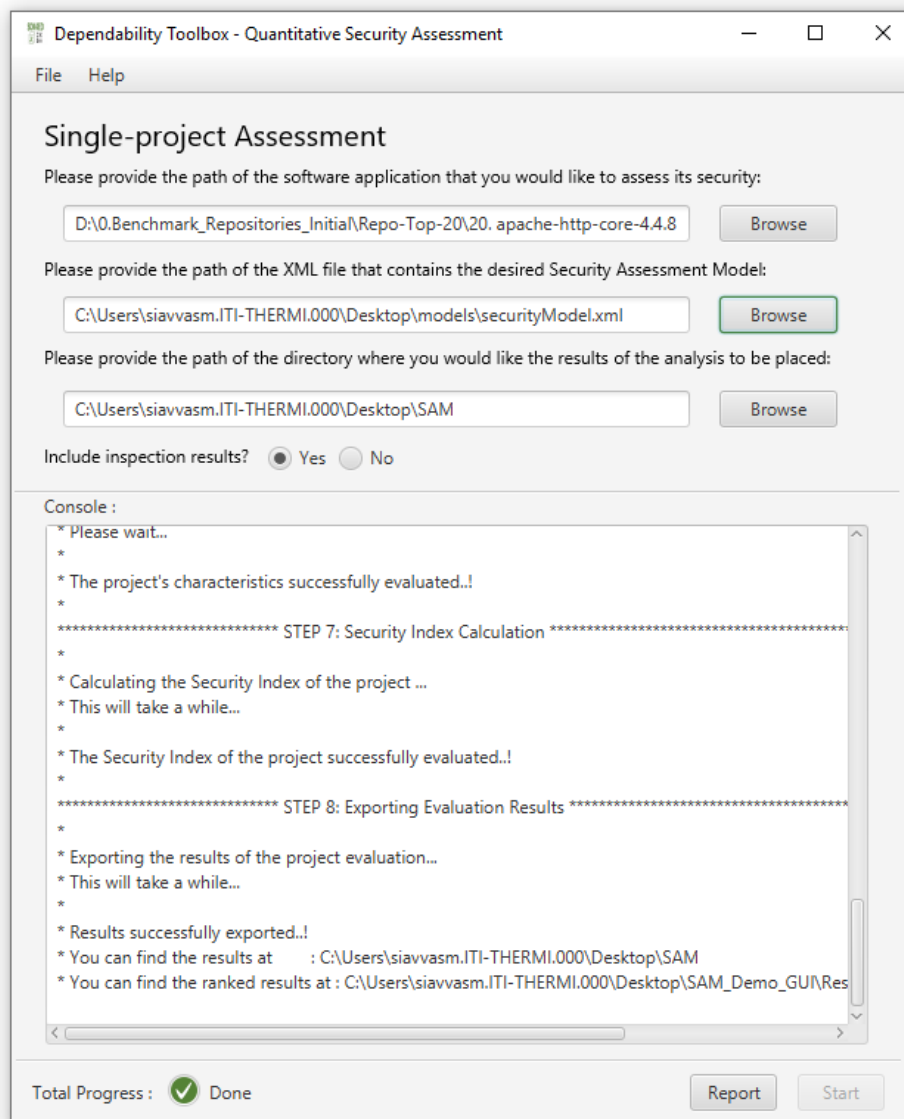


Figure 4.6: The graphic user interface of the offline version of the proposed security assessment model.

The proposed SAM has been developed within the context of the SDK4ED³ project, which is an ongoing EU-funded project. The main purpose of the SDK4ED project is to develop novel indicators able to quantify important quality attributes (i.e., non-functional requirements) of software products, particularly *Energy*, *Maintainability*, and *Security*. The ultimate goal of the project is to provide a CASE platform that will facilitate the development of high-quality software, by identifying potential interrelationships and trade-offs between the often conflicting factors of *Energy*, *Maintainability*, and *Security*, providing also valuable recommendations to the developers on how to improve them. This platform is expected to act as a decision support system that will help developers and project managers monitor the quality of a software product (with respect to these three quality attributes) during the overall development cycle and make more informed decisions regarding potential refactoring and fortification efforts. In fact, the proposed SAM will be integrated into the final SDK4ED platform, in order to be used as the main indicator of *Software Security*. Figure 4.7 depicts the Security Dashboard of the envisaged SDK4ED platform, which is based on the results produced by the proposed SAM.

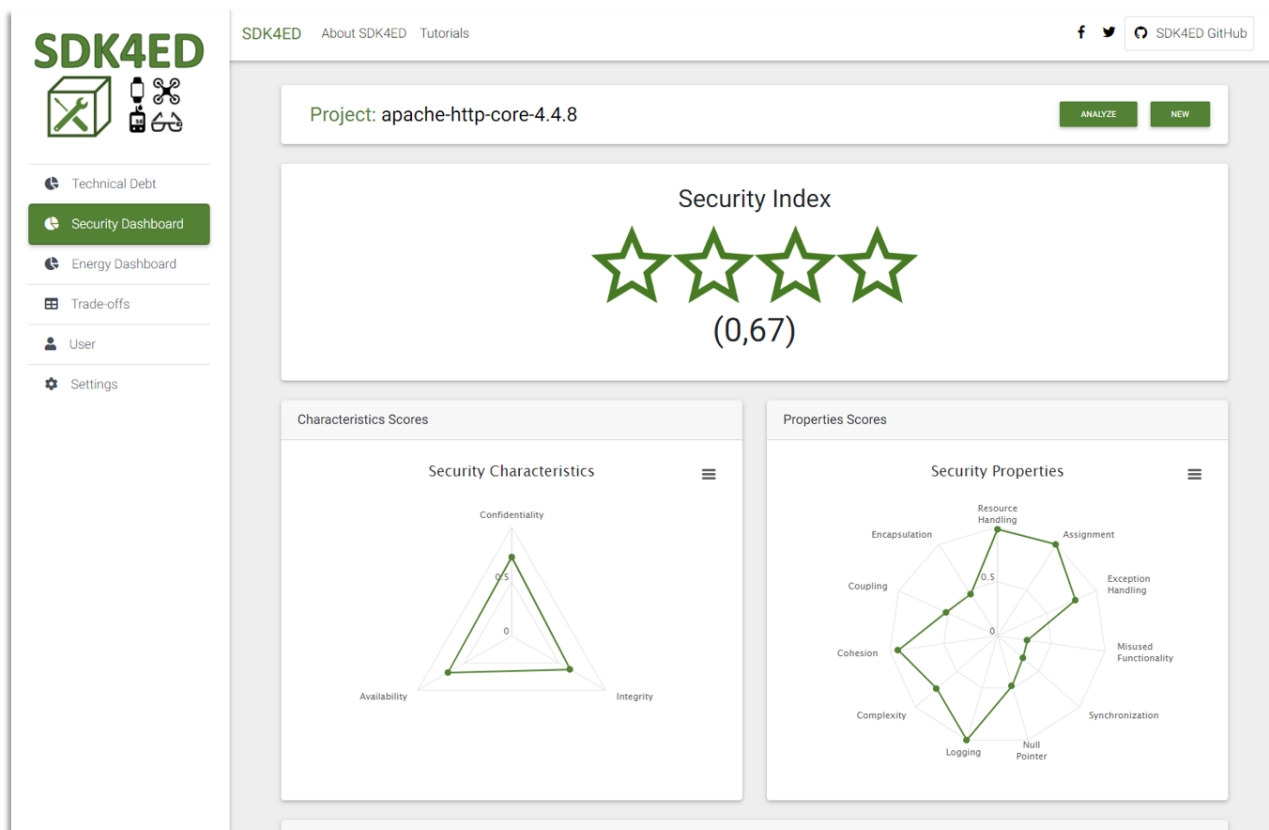


Figure 4.7: The Security Dashboard of the SDK4ED platform.

³<https://sdk4ed.eu/>

4.3 Model Evaluation and Discussion

Contrary to previous research endeavors, which completely lack empirical evaluation of their models or they base their evaluation on a limited number of simple case studies, in the present work we put significant emphasis on the elaborate empirical evaluation of our model since we believe that it is highly required for ensuring its reliability. More specifically, in order to evaluate the correctness of the proposed Security Assessment Model (SAM), as well as to identify its strengths and weaknesses, four individual experiments were conducted, which were based on a large volume of empirical data. The aforementioned experiments along with their corresponding results are reported in the rest of this section.

4.3.1 Comparison with an Independent Security Evaluation Approach

To evaluate the ability of the proposed SAM to reliably assess the internal security level of software products, a comparison of its assessment results with the corresponding results provided by similar models or security experts is considered necessary. However, as already mentioned, no other well-accepted and operationalized security assessment model exists in the literature, while, contrary to software quality [80,81,83], no expert-based security evaluations of software products are publicly available, in order to be used as the reference point of our comparisons. For this purpose, we decided to compare our model to another indicator of the internal security level of software products, namely the *Static Analysis Vulnerability Density (SAVD)* [65].

SAVD, which is the total number of security-related static analysis alerts of a given software application per thousand lines of code, is commonly used in the literature as an indicator of internal software security (see Section 2.1.2). For the quantification of the *SAVD*, the FindBugs [167] static code analyzer was employed, which is a popular open-source static analysis tool for Java, widely used in the literature for security auditing purposes [39,168]. We configured the tool appropriately in order to detect only security-relevant weakness categories including: *Performance*, *Malicious Code*, *Multithreaded Correctness*, and *Security*. The latter bug category is provided by FindSecurityBugs⁴, which is a popular FindBugs plugin.

⁴<https://find-sec-bugs.github.io/>

For the purposes of the experiment, a large code repository of real-world software applications was constructed. In particular, we mined the online GitHub repository and downloaded a large number of open-source Java applications based on their popularity (i.e., number of GitHub stars). From these applications, we kept only those that were able to compile successfully with Maven without any errors or warnings. This process resulted in a relatively large benchmark repository of 150 open-source software applications, comprising approximately 13 million lines of code (as reported by CKJM Extended tool [154]). From the final repository we made sure to exclude software products that were also part of the repository that was used for the calibration of the model (see Section 4.1.2.1).

Subsequently, we analyzed the software applications of the resulting repository using our proposed SAM, in order to calculate their *Security Indexes*. We also applied the FindBugs static code analyzer, in order to calculate the *SAVD* of each one of these applications. Next, we obtained two individual rankings of the analyzed software products, one based on their *Security Indexes*, and another one based on their *SAVDs*. The analyzed software applications, along with their *Security Indexes*, their *SAVDs*, and their corresponding rankings are presented in Table 4.11.

Table 4.11: The security evaluation results of the selected open-source software products, as produced by the proposed security assessment model (i.e., SI), the FindBugs static code analyzer (i.e., SAVD), and a security model that completely omits software metrics (i.e., SI'). Their corresponding rankings are also presented.

Project Name	SI	SI Rank	SAVD	SAVD Rank	SI' Rank	SI' Rank
actframework	0.352024	133	2.363677	71	0.375633	125
ansj_seg	0.457336	96	2.459829	74	0.431643	100
apk-parser	0.583697	54	3.266809	95	0.557523	63
AutoLoadCache	0.341156	136	6.531731	124	0.301215	136
awaitility	0.454287	98	2.30052	68	0.432813	98
azure-sdk-for-java	0.478949	91	6.849798	127	0.529085	68
Cerberus	0.196344	148	16.933207	146	0.119288	148
citrus	0.470902	93	3.290582	96	0.462166	93
clarity	0.565259	57	3.892162	104	0.569655	58
cloudhopper-smpp	0.229077	147	5.207646	118	0.202855	146
commonmark-java	0.600761	48	1.976895	53	0.590902	52
datastructure	0.342878	135	5.310971	119	0.270288	138
dcm4che	0.442593	106	2.587713	78	0.482294	83
disconf-demos-java	0.448576	101	14.954486	144	0.410673	110
dockerfile-maven	0.508837	73	2.866502	85	0.476281	86
docker-java	0.431021	113	6.938421	128	0.420548	107
docker-maven-plugin	0.343779	134	2.133495	62	0.334386	133
docx4j	0.635219	40	3.076223	90	0.682077	36
easy-rules	0.402111	123	4.055343	106	0.35223	129
ECharts	0.545777	60	3.415477	100	0.578902	57
effective-java-examples	0.500633	77	3.989102	105	0.464882	90

Table 4.11 – continued from previous page

Project Name	SI	SI Rank	SAVD	SAVD Rank	SI' Rank	SI' Rank
embedded-redis	0.384162	127	2.555366	77	0.339201	131
EMV-NFC-Paycard-Enrollment	0.897857	12	1.010729	21	0.977198	11
essentials	0.509864	72	2.356457	70	0.479288	84
ews-java-api	0.493813	82	2.04748	56	0.529508	67
Examination _{system}	0.442733	105	5.928509	122	0.429612	102
fastdfs-client-java	0.485265	88	3.309291	98	0.484723	81
fast-serialization	0.776316	26	1.645736	43	0.826841	26
fastweixin	0.48152	90	4.183605	109	0.476895	85
flexmark-java	0.924524	4	1.489728	35	0.994495	5
flip-tables	0.371667	129	4.132231	107	0.283944	137
FLogger	0.312713	140	8.154188	132	0.243971	141
fluent-validator	0.264751	142	8.567662	134	0.214544	144
flyingsaucer	0.888072	13	0.576908	11	0.986969	8
fnlp	0.441396	108	4.466051	114	0.409936	111
gecco	0.257109	144	7.309941	130	0.202556	147
geofire-java	0.918956	5	1.453045	33	0.982995	9
geohash-java	0.714379	34	0.8663	18	0.71695	33
google-oauth-java-client	0.533626	68	2.931691	87	0.510792	70
hbc	0.410178	120	4.194535	110	0.384466	120
Hive2Hive	0.483311	89	10.206814	138	0.462735	91
HotswapAgent	0.390755	124	1.533333	39	0.381015	123
incubator-dubbo	0.341017	137	1.766615	47	0.317506	135
infinittest	0.902785	6	0.856664	17	0.998669	2
itchat4j	0.415945	118	8.811681	135	0.408676	113
itextpdf	0.592183	52	1.010885	22	0.649083	41
J2V8	0.632275	42	2.066902	59	0.671959	38
jade4j	0.902218	8	0.103928	4	0.994247	6
jansi	0.598783	49	1.061445	24	0.62205	43
java-client-api	0.523719	70	1.352874	31	0.524074	69
java-dns-cache-manipulator	0.526045	69	3.818251	102	0.487677	80
javaewah	0.699787	35	0.084814	2	0.736077	32
java-faker	0.569881	56	1.20279	28	0.594224	51
java-memcached-client	0.595624	50	2.066252	58	0.612473	46
java-speech-api	0.485521	87	1.528013	37	0.475346	87
jBCrypt	0.967839	1	0	1	0.977318	10
jcabi-aspects	0.845957	17	0.798722	15	0.860987	20
jdeb	0.541412	64	3.133351	92	0.53169	66
jdonframework	0.102865	150	3.39736	99	0.026705	150
jesque	0.556685	58	1.728866	46	0.566764	59
jgit-cookbook	0.860868	15	1.498011	36	0.881539	18
jHiccup	0.493888	81	4.225352	111	0.451041	95
jieba-analysis	0.58456	53	2.751977	80	0.563133	60
jitwatch	0.544635	61	2.836547	83	0.561145	61
jmustache	0.263917	143	2.060439	57	0.204908	145
jpinyin	0.72193	33	2.19106	63	0.7143	35
json-simple	0.777122	25	0.485358	9	0.771248	29
JSqlParser	0.771822	28	0.249231	7	0.84818	24
Jupiter	0.434233	112	44.267632	149	0.416996	108
jvmtop	0.273965	141	4.157339	108	0.246761	140
jzmq	0.358069	132	2.276539	66	0.351583	130
kilim	0.79681	23	0.90467	20	0.851078	22
kryonet	0.856678	16	0.64683	13	0.912274	15
lanproxy	0.491656	86	8.538633	133	0.454822	94
lanterna	0.495402	79	1.59815	40	0.493085	77
lavagna	0.506193	75	6.435867	123	0.501638	74

Table 4.11 – continued from previous page

Project Name	SI	SI Rank	SAVD	SAVD Rank	SI' Rank	SI' Rank
librec	0.880626	14	1.079325	26	0.944146	14
light-admin	0.534388	67	3.291401	97	0.506519	73
light-task-scheduler	0.45567	97	2.958475	88	0.434122	97
LinuxJavaFixes	0.82487	21	1.060178	23	0.88476	17
Luyten	0.505584	76	1.997624	54	0.508273	72
mango	0.419876	116	2.792263	81	0.395309	117
marytts	0.553067	59	3.772219	101	0.585807	54
metadata-extractor	0.730007	31	2.363935	72	0.747254	31
Minim	0.817954	22	1.106283	27	0.855763	21
mockserver	0.477356	92	2.395899	73	0.472208	88
mp3agic	0.644456	38	1.633259	42	0.662067	39
mybatis-3	0.253467	145	1.813956	48	0.222361	142
mysql-binlog-connector-java	0.60821	46	2.261015	65	0.629682	42
nanohttpd	0.424165	115	3.051317	89	0.414694	109
natty	0.902403	7	0.097171	3	0.998311	3
nlp-lang	0.593405	51	2.492794	75	0.601654	50
obd-java-api	0.662797	36	1.368791	32	0.674689	37
OpenID-Connect-Java-Spring-Server	0.442037	107	6.795628	126	0.447669	96
openmessaging-java	0.534546	66	2.657807	79	0.494144	75
open-replicator	0.784442	24	1.528384	38	0.824081	27
opentracing-java	0.575996	55	2.866356	84	0.557547	62
openwebflow	0.935827	3	0.542672	10	0.999833	1
opsu	0.126092	149	2.278014	67	0.072071	149
ormlite-core	0.362458	131	1.941268	52	0.367949	126
oshi	0.622193	44	3.215105	94	0.609444	49
paoding-rose	0.454234	99	8.962532	136	0.43269	99
paraltec	0.329874	138	10.328638	139	0.321387	134
pcap4j	0.541731	63	1.244906	29	0.583662	55
pcollections	0.843381	18	2.08717	61	0.867907	19
pf4j	0.402288	122	13.391826	143	0.387875	119
pollexor	0.60369	47	2.079002	60	0.611502	48
prettytime	0.946105	2	0.640113	12	0.944575	13
psi-probe	0.42465	114	12.625706	142	0.381166	122
pushy	0.49364	83	157.889437	150	0.484165	82
qart4j	0.767537	29	3.12256	91	0.769518	30
QRGen	0.643101	39	0.199044	6	0.61211	47
re2j	0.90167	10	0.119374	5	0.949749	12
red5-server	0.386592	126	18.724715	147	0.36188	127
RedisClient	0.901686	9	0.468268	8	0.997421	4
restcountries	0.409211	121	24.830699	148	0.354596	128
restx	0.52012	71	2.913895	86	0.510085	71
Resty	0.449362	100	4.437426	113	0.430926	101
rome	0.773322	27	2.310704	69	0.849344	23
Saturn	0.389488	125	7.472227	131	0.396584	116
sentry-java	0.412249	119	4.632893	116	0.383335	121
signpost	0.321402	139	2.228976	64	0.25906	139
simplenlg	0.500524	78	1.477832	34	0.493679	76
smart	0.46726	94	11.790393	141	0.427343	103
spring-roo	0.53518	65	1.667379	45	0.580025	56
sqlite-jdbc	0.493073	85	2.813307	82	0.531702	65
stateless4j	0.494501	80	1.828153	49	0.488287	79
swagger-core	0.493445	84	2.026295	55	0.492698	78
tabula-java	0.631983	43	1.864677	51	0.619789	44
takes	0.829594	20	0.831699	16	0.8341	25
tess4j	0.374361	128	6.786662	125	0.338592	132

Table 4.11 – continued from previous page

Project Name	SI	SI Rank	SAVD	SAVD Rank	SI' Rank	SI' Rank
thymeleaf	0.737764	30	1.075142	25	0.790333	28
traccar	0.63278	41	0.872846	19	0.615746	45
transmittable-thread-local	0.506995	74	4.662477	117	0.470369	89
ttorrent	0.436991	111	9.18105	137	0.423953	104
unirest-java	0.466348	95	1.658833	44	0.423361	105
vlcj	0.368498	130	7.18059	129	0.391746	118
vraptor4	0.419702	117	5.401097	120	0.378306	124
webcam-capture	0.253044	146	5.551322	121	0.217595	143
webdrivermanager	0.440957	109	16.768977	145	0.46272	92
webmagic	0.447593	103	4.351055	112	0.421812	106
weixin4j	0.54181	62	4.575237	115	0.54323	64
weixin-popular	0.443848	104	3.187919	93	0.409644	112
weixin-sdk	0.437277	110	11.148431	140	0.404992	114
Word2VEC _{java}	0.615732	45	1.852995	50	0.586055	53
XChart	0.651997	37	2.503507	76	0.652501	40
xmemcached	0.831587	19	1.609042	41	0.907732	16
ysoserial	0.44773	102	3.818674	103	0.398239	115
ZhiHuSpider	0.900138	11	1.334701	30	0.987149	7
zxing	0.722341	32	0.716788	14	0.715144	34

In order for the *Security Index* to be considered a reliable indicator of software security, a statistically significant positive correlation should be observed between the two rankings. For this purpose, we decided to use the *Spearman's rank correlation coefficient* (ρ) [158], which is a non-parametric and non-sensitive to outliers statistical test. To interpret the strength of the observed correlation, similarly to Chapter 3, the thresholds proposed by Cohen et al. [159] were used. According to Cohen et al. [159], a correlation less than 0.3 is considered weak, between 0.3 and 0.5 is considered moderate, and above 0.5 is considered strong. In general, a positive and close to one correlation denotes that the studied rankings are almost identical. However, it should be noted that in the present experiment we do not expect the rankings to be identical, for reasons that are discussed later in the text.

In order to reach safer conclusions regarding the statistical significance of the observed correlation, we formulated the following null hypothesis (along with its corresponding alternative hypothesis):

H_0 : No statistically significant correlation is observed between the two rankings.

H_1 : A statistically significant correlation is observed between the two rankings.

which was tested at the 95% confidence level ($\alpha = 0.05$).

The calculated *Spearman's rank correlation coefficient* between the two rankings was found to be $\rho = 0.7024$, which is a positive and strong (according to Cohen et al. [159]) correlation. In addition, since the *p-value* was found to be lower than the 0.05 threshold (in fact, *p-value* $< 2.2 \times 10^{-16}$), the null hypothesis was rejected, which led us to the acceptance of the alternative hypothesis, i.e., that a statistically significant correlation exists between the two rankings. This suggests that the two rankings are highly consistent, and, in turn, that the *Security Indexes* produced by our proposed SAM are closely related to their computed *SAVDs*. This denotes that the assessment results provided by our proposed model are closely related to the assessment results produced by another independent security evaluation approach, providing support for its ability to reliably reflect the security level of software products. Hence, the proposed SAM may reliably be used in practice for assessing the security level of software products.

At this point, it should be noted that we did not expect the two rankings to be identical, since the two approaches capture different security-relevant information. More specifically, contrary to *SAVD* that is a coarse-grained indicator of internal software security, our proposed model provides finer-grained assessments, since it groups the security-relevant static analysis alerts into vulnerability categories and calculates an individual *SAVD* for each one of these categories. In addition, it considers security information retrieved from software metrics. Hence, since our proposed SAM uses more security-relevant information for determining the final security score of software products than the *SAVD* metric, the assessment results of the two approaches are not expected to perfectly match.

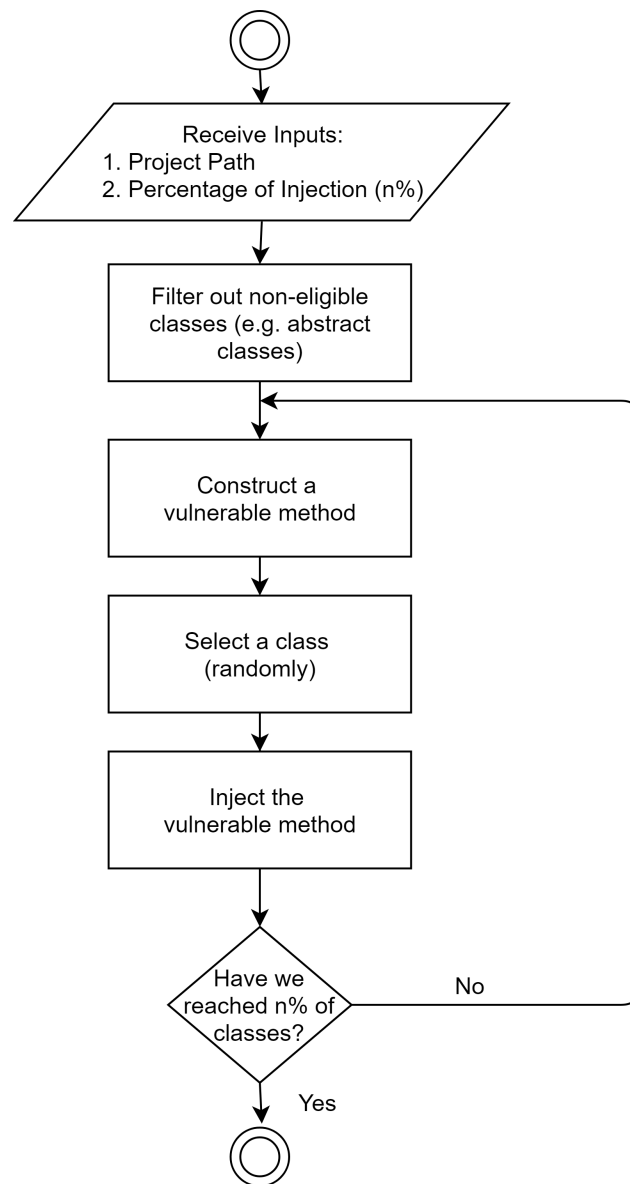
4.3.2 Discretion Power of the Model

The main objective of the proposed SAM is to provide a high-level indicator of software security, in order to facilitate decision making during software development. As already mentioned, the proposed SAM starts from raw results (i.e., security-relevant static analysis alerts and software metrics) and systematically aggregates them in order to produce a single security score (i.e., the *Security Index*), which is obviously more intuitive and easily understandable than the raw static analysis results. However, the main challenge of the model is to avoid masking underlying issues, which may be caused by the many levels of aggregation that are performed for the calculation of the final security score. Hence, in order for the model to be reliable, the introduction of even a trivial number of security-relevant issues between two versions of a software product should

be reflected as a drop in the produced *Security Index*. To this end, the purpose of the present experiment is to evaluate the discretion power of the model, i.e., its ability to detect even a small number of newly introduced security issues.

For the purposes of the experiment, we implemented a dedicated vulnerability injector, i.e., a tool able to inject security-relevant issues (i.e., potential vulnerabilities) to the source code of a given software application written in Java programming language. The main requirements of the produced vulnerability injector were (i) to perform the error seeding in a random manner in order to avoid potential bias, and (ii) to inject security issues that the proposed model is able to detect. The latter is necessary because, as already mentioned, the purpose of the experiment is to evaluate whether the raw low-level issues that the proposed model is able to detect are reflected in the final *Security Index*, or masked by the applied aggregation procedure. Hence, injecting issues that the model is unable to detect, would lead to misleading results. Table 4.12 presents the security-relevant issues that the injector is able to inject, providing their corresponding CWE ID and title.

The overall process that is followed by the vulnerability injector is illustrated in Figure 4.8 in the form of a flow chart diagram. Initially, the injector receives from the user the path of the desired application and the percentage of its classes ($n\%$) that should be injected with potential vulnerabilities. Subsequently, it removes non-eligible classes (e.g., abstract classes, interfaces, etc.), which cannot be injected, and creates a list with the eligible ones. It then constructs a vulnerable method by randomly selecting a number of vulnerability issues from Table 4.12. An example of such a method is presented in Listing 4.1. Then it randomly selects one of the eligible classes and inserts in it the vulnerable method. It also inserts a call to this vulnerable method in one of its methods after excluding getters, setters, and constructors. The same procedure is applied until the desired number of classes is reached (i.e., $n\%$ of the eligible ones).

Figure 4.8: The flow chart of the *Vulnerability Injector*Table 4.12: The list of security issues (i.e., potential vulnerabilities) that the *Vulnerability Injector* is currently able to inject

CWE Entry

CWE-396: Declaration of Catch for Generic Exception

CWE-221: Information Loss or Omission

CWE-209: Information Exposure Through an Error Message

CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion')

CWE-485: Insufficient Encapsulation

CWE-572: Call to Thread run() instead of start()

CWE-476: NULL Pointer Dereference

Listing 4.1: An example of a vulnerable method produced by the *Vulnerability Injector*.

```

public void badMethod() {
    /* Import a file that doesn't exist to cause an exception */
    try {
        FileReader fileReader = new FileReader("path/to/file");
    } catch (Exception e) { /* CWE-221: Information Loss or Omission */
        e.printStackTrace(); /* CWE-209: Information Exposure Through an
                               Error Message */
    }
}

```

For the purposes of the present experiment, the same benchmark repository of real-world software applications that was used in the previous analysis, comprising 150 popular open-source Java applications was utilized. Based on this repository we performed 3 sets of vulnerability injections. More specifically, we injected vulnerabilities to the 1%, 10%, and 25% of the eligible classes of the software applications which reside in the benchmark repository. After each injection, the whole repository was recompiled in order to produce the binaries (i.e., class files), which were necessary for the assessment. Only those applications that were able to compile successfully without additional warnings or errors were selected for the final experiment. In fact, from the 150 software applications of the benchmark repository, the 131 of them were able to compile correctly without additional issues. The injected applications were then analyzed using our proposed SAM, and their *Security Indexes* are presented in Table 4.13.

Table 4.13: The *Security Indexes* of open-source applications after going through vulnerability injection.

Project Name	<i>SI_{original}</i>	<i>SI_{1%}</i>	<i>SI_{10%}</i>	<i>SI_{25%}</i>
actframework	0.352024	0.343967	0.251141	0.187745
ansj_seg	0.457336	0.443769	0.344782	0.280851
apk-parser	0.583697	0.542649	0.370576	0.278384
AutoLoadCache	0.341156	0.330022	0.279603	0.221497
awaitility	0.454287	0.454287	0.317471	0.259945
azure-sdk-for-java	0.478949	0.456946	0.35437	0.292814
Cerberus	0.196344	0.116318	0.101777	0.090612
citrus	0.470902	0.445511	0.320862	0.230665
clarity	0.565259	0.545041	0.417213	0.318574
cloudhopper-smpp	0.229077	0.227432	0.197935	0.143849
commonmark-java	0.600761	0.568416	0.414812	0.284559
datastructure	0.342878	0.338122	0.307342	0.241162
dcm4che	0.442593	0.436778	0.397911	0.336393
disconf-demos-java	0.448576	0.370105	0.24512	0.163774
dockerfile-maven	0.508837	0.390313	0.390313	0.290177
docker-java	0.431021	0.382882	0.221588	0.152934
docker-maven-plugin	0.343779	0.32669	0.265841	0.183987

Table 4.13 – continued from previous page

Project Name	$SI_{original}$	$SI_{1\%}$	$SI_{10\%}$	$SI_{25\%}$
docx4j	0.635219	0.61829	0.515065	0.416619
easy-rules	0.402111	0.362793	0.222248	0.169057
ECharts	0.545777	0.521376	0.360537	0.25602
effective-java-examples	0.500633	0.463412	0.344919	0.285154
embedded-redis	0.384162	0.296804	0.247277	0.229425
essentials	0.509864	0.498703	0.469617	0.451142
ews-java-api	0.493813	0.480297	0.407534	0.321918
Examination _{system}	0.442733	0.442733	0.328905	0.295759
fastdfs-client-java	0.485265	0.485265	0.399554	0.314383
fast-serialization	0.776316	0.761969	0.738295	0.693991
fastweixin	0.48152	0.459449	0.316142	0.25164
flexmark-java	0.924524	0.924496	0.915084	0.882224
fluent-validator	0.264751	0.211517	0.104249	0.073031
fnlp	0.441396	0.427898	0.369922	0.291956
gecco	0.257109	0.242501	0.173926	0.1267
geohash-java	0.714379	0.554184	0.474674	0.329395
google-oauth-java-client	0.533626	0.456387	0.279106	0.207563
hbc	0.410178	0.380124	0.250954	0.188347
Hive2Hive	0.483311	0.483311	0.332146	0.250775
HotswapAgent	0.390755	0.386231	0.32558	0.262075
incubator-dubbo	0.341017	0.328691	0.265764	0.23135
itchat4j	0.415945	0.395977	0.374598	0.348051
itextpdf	0.592183	0.583421	0.517781	0.433613
J2V8	0.632275	0.620926	0.532211	0.442839
jade4j	0.902218	0.902171	0.894924	0.847366
jansi	0.598783	0.598539	0.530301	0.503714
java-client-api	0.523719	0.454379	0.311415	0.237173
javaewah	0.699787	0.69978	0.537004	0.44253
java-faker	0.569881	0.569881	0.304856	0.240969
java-speech-api	0.485521	0.485521	0.410912	0.38181
jcabi-aspects	0.845957	0.759561	0.661231	0.53554
jdeb	0.541412	0.509106	0.394245	0.312157
jdonframework	0.102865	0.093311	0.085093	0.082731
jesque	0.556685	0.556465	0.533378	0.488221
jgit-cookbook	0.860868	0.835267	0.704882	0.577861
jieba-analysis	0.58456	0.488172	0.487794	0.426844
jitwatch	0.544635	0.538237	0.469515	0.401801
jmustache	0.263917	0.169661	0.141551	0.091969
JSqlParser	0.771822	0.764178	0.641924	0.528942
Jupiter	0.434233	0.424789	0.355087	0.320407
jvmtop	0.273965	0.265731	0.264068	0.251226
jzmq	0.358069	0.357702	0.330382	0.276126
kilim	0.79681	0.793863	0.760818	0.713261
kryonet	0.856678	0.843191	0.836679	0.817007
lanproxy	0.491656	0.472639	0.438913	0.371588
lanterna	0.495402	0.484513	0.429778	0.384932
lavagna	0.506193	0.484234	0.40415	0.324363
librec	0.880626	0.87893	0.849054	0.814136
light-admin	0.534388	0.454951	0.288249	0.178013
light-task-scheduler	0.45567	0.434087	0.364907	0.302089
LinuxJavaFixes	0.82487	0.817212	0.817209	0.811357
Luyten	0.505584	0.491516	0.463108	0.40396
mango	0.419876	0.392812	0.266014	0.189145
marytts	0.553067	0.548324	0.509697	0.461853
metadata-extractor	0.730007	0.697415	0.560236	0.432206

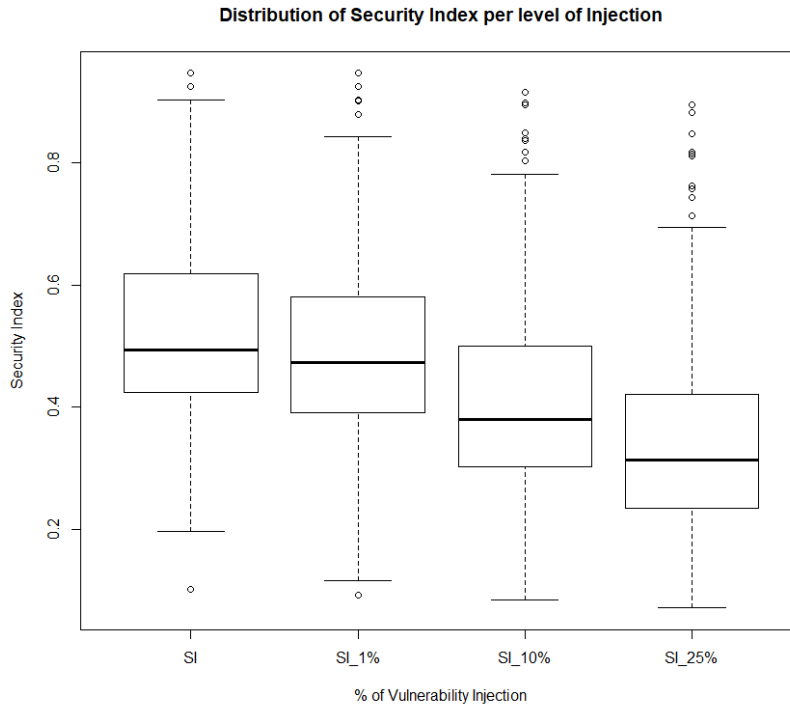
Table 4.13 – continued from previous page

Project Name	$SI_{original}$	$SI_{1\%}$	$SI_{10\%}$	$SI_{25\%}$
Minim	0.817954	0.811651	0.781035	0.743482
mockserver	0.477356	0.465706	0.379591	0.31669
mp3agic	0.644456	0.643888	0.507346	0.41413
mybatis-3	0.253467	0.219222	0.119903	0.071508
mysql-binlog-connector-java	0.60821	0.588843	0.496418	0.392033
nanohttpd	0.424165	0.42384	0.377409	0.311873
nlp-lang	0.593405	0.577926	0.519929	0.432324
obd-java-api	0.662797	0.566094	0.404516	0.281353
OpenID-Connect-Java-Spring-Server	0.442037	0.434625	0.329451	0.258615
openmessaging-java	0.534546	0.426148	0.382544	0.321658
open-replicator	0.784442	0.760755	0.659755	0.550302
opentracing-java	0.575996	0.533617	0.454393	0.387832
ormlite-core	0.362458	0.35418	0.27467	0.201593
oshi	0.622193	0.60143	0.509096	0.400173
paoding-rose	0.454234	0.44441	0.376851	0.313296
paraltec	0.329874	0.319669	0.266816	0.242659
pcap4j	0.541731	0.533679	0.482257	0.415653
pcollections	0.843381	0.78003	0.715853	0.672371
pf4j	0.402288	0.37047	0.291384	0.222261
prettytime	0.946105	0.946105	0.839409	0.756861
psi-probe	0.42465	0.417118	0.331299	0.27199
pushy	0.49364	0.493594	0.455646	0.366757
qart4j	0.767537	0.689786	0.569791	0.516853
QRGen	0.643101	0.531669	0.400739	0.336487
red5-server	0.386592	0.378048	0.324807	0.275212
RedisClient	0.901686	0.901678	0.897839	0.894393
restcountries	0.409211	0.346277	0.301389	0.201738
restx	0.52012	0.50839	0.418083	0.355805
Resty	0.449362	0.426592	0.315516	0.239358
rome	0.773322	0.764871	0.700611	0.636363
Saturn	0.389488	0.384044	0.32785	0.281945
sentry-java	0.412249	0.403277	0.364919	0.337222
signpost	0.321402	0.293694	0.217543	0.182213
smart	0.46726	0.422626	0.290259	0.199294
spring-roo	0.53518	0.522306	0.458172	0.362744
sqlite-jdbc	0.493073	0.482678	0.462218	0.433642
stateless4j	0.494501	0.494501	0.329849	0.231485
swagger-core	0.493445	0.473674	0.282874	0.23113
tabula-java	0.631983	0.592542	0.494188	0.475451
takes	0.829594	0.793471	0.629119	0.499926
tess4j	0.374361	0.374361	0.353686	0.267663
thymeleaf	0.737764	0.711179	0.584382	0.468349
traccar	0.63278	0.599191	0.484344	0.387687
transmittable-thread-local	0.506995	0.468214	0.424198	0.38053
torrent	0.436991	0.417945	0.380512	0.319233
unirest-java	0.466348	0.375272	0.312737	0.240977
vlcj	0.368498	0.345768	0.198761	0.14175
vraptor4	0.419702	0.378472	0.159071	0.085575
webcam-capture	0.253044	0.245153	0.199477	0.154831
webdrivermanager	0.440957	0.43232	0.335839	0.292603
webmagic	0.447593	0.416051	0.273064	0.214924
weixin4j	0.54181	0.513727	0.37923	0.274158
weixin-popular	0.443848	0.393402	0.211224	0.150336
weixin-sdk	0.437277	0.401593	0.279026	0.22022
Word2VEC _{java}	0.615732	0.502089	0.501971	0.367887

Table 4.13 – continued from previous page

Project Name	$SI_{original}$	$SI_{1\%}$	$SI_{10\%}$	$SI_{25\%}$
XChart	0.651997	0.627409	0.491825	0.373863
xmemcached	0.831587	0.83015	0.80285	0.762659
ysoserial	0.44773	0.439218	0.405839	0.347767
zxing	0.722341	0.707105	0.606375	0.502947

As can be seen by Table 4.13, the security scores of the injected software applications tend to be smaller compared to the corresponding scores of their original versions. This is observed even in the case of 1% injection. Another interesting observation is that the security scores of the analyzed software applications seem to follow a declining trend with respect to the level of injection. In fact, the higher the level of injection, the lower the *Security Index* of the software products. (i.e., the more the vulnerabilities, the lower the security score). This can be observed more clearly in the box plots presented in Figure 4.9. In fact, the 1%, 10%, and 25% injection led to an average reduction in the *Security Indexes* of the analyzed software applications by 5.66%, 22.72%, and 35.96% respectively. This suggests that even the introduction of a small number of security issues leads to a relatively high reduction in the produced security score, which is important because even a single vulnerability may lead to far-reaching consequences.

Figure 4.9: The boxplots of the *Security Indexes* of each injection groups.

Formal mathematical analysis is required, in order to reach safer conclusions. For this purpose, we decided to employ hypothesis testing in order to examine whether there are any statistically significant differences between the security scores of the four groups of software applications. If a statistically significant difference exists at least between the security scores of the original applications and those of their injected versions, we can conclude that the model has sufficient discretion power.

The four groups are considered dependent since the same subject (i.e., software application) was involved in each round of vulnerability injection/security evaluation. This indicates that we have a repeated measures experiment, and therefore the repeated-measures analysis of variance (rANOVA) seems a good statistical tool for our experiment. In fact, rANOVA is a statistical test that is used to determine whether statistically significant differences exist between different samples (usually more than two) in a repeated measurement test, i.e., when the samples are dependent.

However, rANOVA is a parametric test, meaning that specific assumptions should be satisfied, in order for the test to provide reliable results. The main assumption that should be satisfied is the *Normality Assumption*. This assumption requires the pairwise differences between the studied groups to follow a normal distribution. In order to test this assumption, we used the *Shapiro-Wilk Test*. We formulated the following null hypothesis (along with its alternative hypothesis):

H_0 : *The difference between the two selected groups follow a normal distribution.*

H_1 : *The difference between the two selected groups does not follow a normal distribution.*

and we tested it at the 95% confidence interval. More specifically, we applied the *Shapiro-Wilk Test* for each one of the possible pairs of the four groups of security scores presented in Table 4.13. The resulting *p-values* of the applied tests are reported in Table 4.14.

As can be seen by Table 4.14, the null hypothesis is rejected for the differences between the security scores of the original applications, and those of the 1% and 10% injection groups, as well as for the pairwise differences between the security scores of latter two groups, indicating that these pairs do not follow a normal distribution. However, the null hypothesis cannot be

Table 4.14: The results (i.e., p -values) of the Shapiro-Wilk Normality Test.

	SI	$SI_{1\%}$	$SI_{10\%}$	$SI_{25\%}$
SI	-	1.2×10^{-12}	0.0052	0.249
$SI_{1\%}$		-	0.0054	0.1608
$SI_{10\%}$			-	0.27
$SI_{25\%}$				-

rejected for the pairwise differences between the scores of the 25% injection group and those of the other three groups.

Hence, for these three cases (i.e., pairwise differences) we also checked the second assumption of rANOVA, which is the *Sphericity Assumption*. The *Sphericity Assumption* requires the variances of the differences between all the possible pairs of the studied samples (i.e., groups) that take part in the experiment to be equal. In order to test this assumption, we employed *Mauchly's test*. More specifically we formulated the following null hypothesis, which we tested at the 95% confidence level:

H_0 : The variances of the differences between all the studied pairs are equal.

H_1 : The variances of the differences between all the studied pairs are not equal.

The probability of the *Mauchly's test* was found to be lower than 2.2×10^{-16} (i.e. $p\text{-value} < 2.2 \times 10^{-16}$), which is significantly smaller than the threshold of 0.05. This led us to the rejection of the null hypothesis, leading to the conclusion that the variances of the studied differences significantly differ. Therefore, this provides strong evidence that the *Sphericity Assumption* is violated.

From the above analysis it is obvious that in the present experiment the two main assumptions of rANOVA are not satisfied. Although rANOVA is considered to be robust to moderate violations of normality and sphericity, in our case, since the violations were found to be statistically significant, it is better to avoid the adoption of rANOVA, and use a non-parametric alternative. To this end, we decided to use *Friedman test*, which is a popular non-parametric alternative to the repeated-measures ANOVA. For this purpose, we formulated the following null hypothesis (along with its alternative hypothesis):

H_0 : No statistically significant differences are observed between the Security Indexes of the studied groups.

H_1 : Statistically significant differences are observed between the Security Indexes of the studied groups.

which was tested at the 0.05 level of confidence. Since the p -value of the test was found to be lower than 2.2×10^{-16} , which is significantly smaller than the threshold of 0.05, the null hypothesis is rejected. Therefore, we can conclude that the security scores of the four studied groups are significantly different.

Finally, in order to specifically determine which of the studied groups are significantly different from the others, post-hoc analysis was performed. For this purpose, we employed the *Nemenyi test*, which is a post-hoc test that is widely used along with the *Friedman test*. The results of the *Nemenyi test* are presented in Table 4.15. As can be seen by Table 4.15, all the calculated p -values are lower than the 0.05 threshold, which denotes that the security scores of each one of the four groups are statistically significantly different from the security scores of the three other groups.

Table 4.15: The results (i.e., p -values) of the post-hoc Nemenyi Test.

	SI	$SI_{1\%}$	$SI_{10\%}$	$SI_{25\%}$
SI	-	2.7×10^{-8}	$< 2.2 \times 10^{-16}$	$< 2.2 \times 10^{-16}$
$SI_{1\%}$		-	7.3×10^{-10}	$< 2.2 \times 10^{-16}$
$SI_{10\%}$			-	1.9×10^{-9}
$SI_{25\%}$				-

From the above analysis, it is obvious that even the injection of a small number of security-relevant issues (i.e., potential vulnerabilities) leads to a clear reduction in the produced *Security Index* of a software application. This suggests that the introduction of even a small number⁵ of security issues is highlighted by the proposed SAM with a relevant drop in the produced *Security Index*. In simple words, the model does not seem to mask existing underlying issues, and therefore we can conclude that it has sufficient discretion power.

⁵It should be noted that even the 25% injection is considered injection of a small number of security issues. 25% injection means that the 25% of the eligible classes of a software application were injected with a small number (usually one or two) of potentially vulnerable instructions, which is trivial compared to the overall size (i.e., lines of code) of the application.

4.3.3 Impact of Software Metrics

As mentioned in Section 4.1.1, we decided to include software metrics in the proposed model, in order to enhance the completeness of the security assessment, by taking into account factors that are indirectly related to software security. However, since the weights of the metric-based properties are determined based on expert judgments, they inevitably include subjectivity, which may potentially affect the correctness (i.e., objectivity) of the overall assessment. Although the total impact of the software metrics on the produced *Security Index* was observed to be small (i.e., approximately 11-12%), in order to justify our choice to include software metrics in the final model a formal evaluation of their actual impact on the overall security assessment is required. For this purpose, the assessment results of the proposed model need to be compared with the corresponding assessment results of an equivalent model that omits software metrics.

To this end, we created an alternative security assessment model that is based exclusively on static analysis-based properties (i.e., vulnerability categories). The thresholds of the static analysis-based properties of the new model are the same with those of the original model (see Table 4.5), while its weights were elicited based on the same pairwise comparison matrices (e.g., Figure 4.5), using only the values that correspond to vulnerability categories. Thus, since the thresholds of the model are data driven and the weights are determined based on information retrieved from the CWE knowledge base (instead of expert judgments), the new model is free from subjectively-defined parameters.

For the purposes of the experiment, we analyzed the same software applications that were used in the previous analyses, i.e., the 150 popular Java applications retrieved from GitHub repository (see Section 4.3.1), using the newly constructed model, and we ranked them based on their computed *Security Indexes*. The *Security Indexes* of the analyzed applications along with their corresponding ranking are presented in Table 4.11. Subsequently, we compared this ranking with the ranking of the software applications obtained based on the *Security Indexes* that were calculated by the original model. For the comparison of the two rankings the *Spearman's Rank Correlation Coefficient* (ρ) [158] was employed, and the following null hypothesis (along with its corresponding alternative hypothesis) was formulated and tested at the 95% ($\alpha = 0.05$) level of confidence:

H_0 : No statistically significant correlation exists between the two rankings.

H_1 : *A statistically significant correlation is observed between the two rankings.*

The calculated ρ between the two rankings was found to be $\rho = 0.989$, which is a correlation highly close to the value of 1. This suggests that the two rankings are almost identical. In addition, the *p-value* of the correlation was found to be highly lower than the threshold of 0.05 (in fact, $p\text{-value} < 2.2 \times 10^{-16}$). This led us to the rejection of the null hypothesis, which indicates that the observed correlation is statistically significant.

From the above analysis, we can conclude that the two models provide almost identical assessment results. In fact, although the security scores that are produced by the original model are slightly different from the corresponding scores produced by the model that omits software metrics (Table 4.11), the overall assessment (i.e., ranking) is almost the same. This indicates that the inclusion of the software metrics enriches the produced security scores with additional security-relevant information without significantly affecting the overall evaluation. This also suggests that the subjective parameters of the proposed model (i.e., expert-defined weights) that were introduced because of the inclusion of the software metrics, do not seem to have a significant impact on the overall assessment, and, in turn, on the objectivity of the final model. Therefore, the two models can be used interchangeably in practice for assessing the internal security of software products. Although we recommend the adoption of the model with the software metrics since it is more complete, we also provide the model that is based only on static analysis-based properties on the web page with the supporting material of the present work [38].

4.3.4 Class-level Assessment and Vulnerability Prediction

The purpose of the previous experiments was to examine the ability of the proposed SAM to reliably evaluate the internal security of complete software products. However, an equally interesting question is whether the proposed model can be used to reliably assess the security level of individual software components (i.e., classes). In other words, we are interested in knowing whether the security score of a single software component (i.e., class) may indicate the existence of software vulnerabilities in the given class. To this end, in this section, we investigate the ability of the proposed SAM to discriminate between vulnerable and clean classes. A positive answer to this question will suggest that the model can provide reliable class-level estimations of

software security and that the produced security score can be used as the basis for vulnerability prediction. Since the produced security score is, in fact, an aggregation of static analysis alerts and software metrics, this will also verify the ability of these factors to be used as vulnerability indicators.

It should be noted that proving the discriminative power of the proposed model and, thus, its ability to be used as the basis for vulnerability prediction is of special interest since existing VPMs are based exclusively on ML algorithms (see Section 2.2.5). This hinders their transparency, as it is not clear how a specific decision was reached by the models. This information would be valuable to detect problematic characteristics of the source code that may contribute to the introduction of vulnerabilities. Hence, a transparent VPM, which allows its predictions to be traced back to its inputs would be valuable. The proposed SAM constitutes a promising candidate for the construction of such a VPM as it is a mathematical and statistical model with a transparent structure that can be easily traced.

4.3.4.1 Vulnerability Dataset

For the purposes of the present experiment, a highly balanced dataset of vulnerable and clean software components is required. However, current literature lacks such a reliable dataset. In fact, as stated in Section 2.2.5, existing datasets are highly unbalanced, while they are constructed based mainly on reported vulnerabilities of software products (e.g., [106]). This affects their correctness, since not all of the vulnerabilities that a product contains are always reported, and therefore many components that are considered clean in the dataset may, in fact, be vulnerable, as they may contain vulnerabilities that have not been reported yet. Hence, for the needs of the present study, and in order to enhance the reliability of its results, a highly balanced dataset of clean and vulnerable components was constructed based on the OWASP Benchmark [121].

OWASP Benchmark [121] is a popular test suite that is commonly used for the evaluation of static code analyzers regarding their ability to detect vulnerabilities. It is a collection of a large number of software components that contain known vulnerabilities. The reason for selecting this benchmark as the basis of our study is twofold. Firstly, the software components provided by the benchmark are Java classes, and therefore they can be easily analyzed by

the proposed SAM. Secondly, contrary to similar test suites (e.g., Juliet Test Suite [119]), the selected benchmark comprises also software components that do not contain actual vulnerabilities (i.e., they contain false positives). In particular, the OWASP Benchmark v1.2 was used which comprises 2740 software components, of which 1415 contain actual vulnerabilities, and 1325 contain false positives. The classes containing actual vulnerabilities were selected as the vulnerable components, while those containing false positives as the clean components of the present analysis.

The proposed SAM was employed in order to calculate the security scores (i.e., the *Security Indexes*) of the selected vulnerable and clean software components. Subsequently, based on these scores duplicates were removed, in order to remove bias imposed by highly similar components. This was necessary since the same component was frequently used by the benchmark for testing different types of vulnerabilities. After eliminating duplicates, the dataset consisted of 1061 vulnerable and 1007 clean classes. In order to construct a highly balanced dataset, 600 observations were randomly selected from each group of vulnerable and clean components. Undersampling was used in order to reduce the possibility of selecting highly common components for the final analysis. Hence, this led to the construction of a highly balanced dataset, comprising 600 vulnerable and 600 clean Java classes.

4.3.4.2 Discriminant Analysis

In order to investigate the ability of the proposed SAM to discriminate between vulnerable and clean software components, statistical analysis was employed. Initially, the box plots of the security scores of the vulnerable and clean components (calculated through the aforementioned process) were plotted, to allow the visual comparison of their distributions (Figure 4.10a).

Figure 4.10a clearly shows that the security scores of vulnerable classes tend to be lower compared to the scores of the clean components. This indicates that the produced *Security Index* may possibly be used as an indicator of vulnerable components. However, in order to reach safer conclusions hypothesis testing was applied. More specifically, *Wilcoxon Rank Sum test* was performed between the security scores of vulnerable and clean software components in order to investigate whether a statistical significant difference exists between their values. *Wilcoxon*

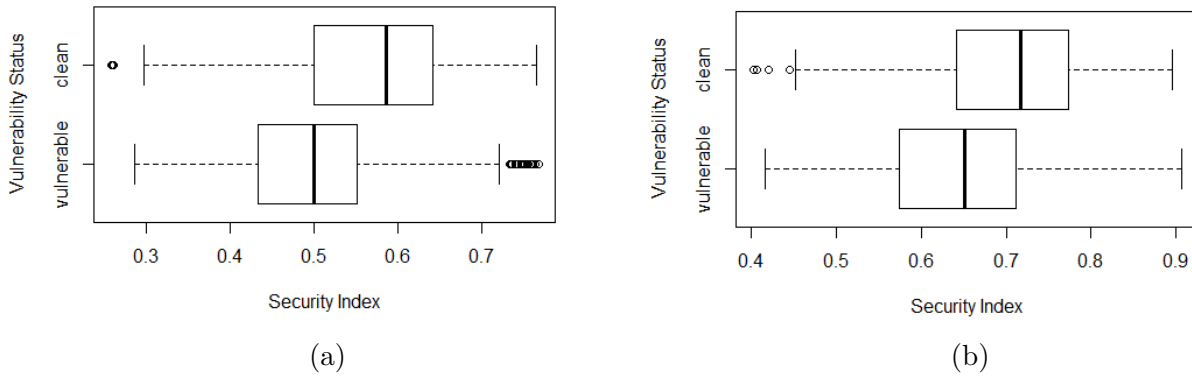


Figure 4.10: Distribution of the security scores of the vulnerable and clean software components of the dataset, as produced by (a) the original model, and (b) the recalibrated model.

Rank Sum test is a non-parametric test, which is not sensitive to outliers and does not assume any distribution for the studied data. It has been widely used in the related literature for testing the ability of different factors to discriminate between vulnerable and clean software artifacts (e.g. [49–51, 110]). In particular, the following null hypothesis (along with its corresponding alternative hypothesis) was formulated and tested with confidence level 95% (i.e., $\alpha = 0.05$):

H_0 : No difference exists between the security scores of vulnerable and clean software components.

H_1 : The security scores of the clean and vulnerable components are statistically different.

After performing the test the *p-value* was found to be much lower than the threshold of 0.05 (i.e. $p\text{-value} < 2.2 \times 10^{-16}$). Hence, the null hypothesis is rejected, leading to the acceptance of the alternative hypothesis, denoting that a statistically significant difference exists between the security scores of the vulnerable and clean components. This suggests that the proposed SAM can discriminate between vulnerable and clean software components, and thus its score can be used as an indicator of vulnerabilities.

The above analysis was replicated by calibrating the security model on the complete dataset of vulnerable and clean components constructed for the purposes of this experiment. The produced model was subsequently applied for assessing the security level of the 600 vulnerable and 600 clean classes of the final dataset. The corresponding distributions of the produced security scores are presented in Figure 4.10b. As can be seen by Figure 4.10b, the security scores of the clean components are again observed to be higher compared to the vulnerable ones. This

observation is supported by the results of *Wilcoxon Rank Sum* test. The *p-value* was found again to be below the value of 0.05, leading to the conclusion that the newly calibrated model is also able to discriminate between vulnerable and clean software components.

In brief, since the model is able to discriminate between vulnerable and clean software classes, we can conclude that it can potentially provide reliable class-level security assessments. In fact, the results of the present experiment lead us to the conclusion that the value of the produced security score can potentially indicate whether the analyzed class contains vulnerabilities or not. This also provides preliminary evidence that the proposed SAM can be used as the basis for the construction of a transparent vulnerability prediction model, able to highlight classes of a software product that are potentially vulnerable. For instance, this could be achieved by training an appropriate threshold for the produced *Security Index*, below which a class will be marked as vulnerable. This constitutes an interesting direction for future research.

Another interesting observation is that since the proposed model is based on static analysis alerts and software metrics, the results of the present experiment provide further support to previous studies regarding the ability of these factors to indicate the existence of software vulnerabilities. More specifically, according to our findings, the combination of static analysis alerts with software metrics may potentially lead to models with sufficient discriminative power.

4.4 Summary and Critical Analysis

In the present section, a fully-automated hierarchical Security Assessment Model (SAM) was proposed that is able to quantify the internal security level of software products written in Java, based on low-level security indicators (i.e., static analysis alerts and software metrics). In brief, the proposed model, following the guidelines of ISO/IEC 25010 [26] and based on a set of thresholds and weights, aggregates these low-level indicators in order to produce a single security score (i.e., the *Security Index*) that reflects the internal security level of the analyzed software. The model was calibrated based on a large code repository comprising 100 popular Java applications retrieved from the Maven Repository [152], as well as on the knowledge retrieved from the Common Weakness Enumeration (CWE) [41]. In order to produce a reliable set of weights that reflect the knowledge expressed by CWE instead of expert judgments, a novel weights elicitation approach grounded on the AHP [162] and SMARTS/SMARTER

[166] decision-making techniques was developed and used. The proposed model was evaluated through a set of experiments that were based on 150 popular open-source Java applications retrieved from GitHub, as well as on 1200 test cases retrieved from the OWASP Benchmark. The results of these experiments showcased the ability of the model to reliably reflect the internal security of software products both at product- and class-level of granularity, with sufficient discretion power. They also provide evidence for its ability to discriminate between vulnerable and clean software components, and therefore to be used as the basis for vulnerability prediction. Since the model is based on static analysis alerts and software metrics, the latter also provides further support regarding the ability of these factors (and especially their combination) to indicate the existence of software vulnerabilities.

It should be noted that the proposed model attempts to satisfy all the requirements for *practicality* and *reliability* that were defined in Section 2.1.4. The model is highly *practical*, since: (i) it provides a high-level quantitative expression of software security, (ii) it is fully-automated, and (iii) it is operationalized in the form of individual tools [38] (see Section 4.2), allowing its adoption in practice. It is also sufficiently *reliable* (compared to its counterparts), since it was constructed based on fundamental concepts retrieved from international standards and also from the mature field of software quality assessment (e.g., ISO/IEC 25010 [26]), while its design parameters (i.e., thresholds and weights) were derived in a way to sufficiently avoid subjective information (see Sections 4.1.2.1 and 4.1.2.2). In addition to this, the proposed model was built and evaluated on a large volume of empirical data (i.e., a broad repository of 250 software applications, comprising approximately 20 million lines of code), which also enhances its reliability. To the best of our knowledge, this is the first fully-automated and operationalized security assessment model that can be found in the related literature, while it is the only model that was built and evaluated on such a large volume of empirical data.

Chapter 5

Optimum Checkpoint Interval: Time Considerations

Cloud and Fog Computing allows diverse software applications to run on complex interconnected systems where reliability and security can be of significant concern. Major failures in such systems occur [171], due to complex effects between various factors including human decisions and systemic interactions in the architecture, the software systems, and the network connections [172]. Furthermore, a recent report [173] states that “*The main problems affecting the cloud are insecure interface APIs, shared resources, data breaches, malicious insiders, and misconfiguration issues*” including active adversarial mechanisms [174]. Clearly, Cloud providers will do their best to improve the security and reliability of their platforms. However, we also need methods that can limit the average execution time of applications that run on the Cloud and Fog despite the intermittent failures of the platforms. This is particularly of interest for long-running applications or those that are run frequently and repeatedly.

One such mechanism that is particularly examined in the present thesis is the Application-level Checkpoint and Restart (ALCR) that is widely used to enhance the reliability of long-running programs [31,33,34] by periodically saving a copy or checkpoint of the current execution state of software. The most recent copy is then used to restart program execution in case of

Part of the work presented in this chapter has appeared in the proceedings of the 6th IEEE International Conference on Cyber Security and Cloud Computing (IEEE CSCloud 2019) [169], as well as in a paper published in the Elsevier’s Simulation Modelling Practice and Theory journal [170].

failure. Originally developed for transaction-oriented systems and databases [35–37, 123, 124], it has been widely adopted to improve the reliability of modern High Performance Computing (HPC) [127, 140] software.

Long intervals of time between checkpoints will increase the overhead associated with system restart, while short intervals will increase the overhead caused by the checkpoints themselves. The checkpoint interval must then be optimized so as to minimize a program’s expected execution time in the presence of failures [175–177]. In [178, 179] the impact of asynchronous checkpointing strategies on the performance of distributed systems has been studied.

Among the existing checkpointing strategies, ALCR [31, 32] uses a small memory footprint [33, 34], but requires significant expertise for the selection of source code locations in which checkpoints should be inserted. Yet existing ALCR tools and libraries facilitate the insertion of checkpoints in the software programs by identifying the most suitable locations in which checkpoints should be inserted. These locations are normally long-running loops, since computational loops constitute a significant source of failure-related re-executions [122, 128]. However such tools do not provide a method to select the inter-checkpoint interval, which has a significant influence on the average execution time of software.

To this end, in the present chapter, we propose that the inter-checkpoint intervals in specific loop be selected optimally as a function of program failure rate, the execution cost for establishing a checkpoint, and the execution time related to restarting the program after a failure, based on a mathematical model. We suggest that this approach can be implemented as an API within an ALCR tool, to select the optimum checkpoint interval in program loops.

The rest of the chapter is structured as follows: Section 5.1 provides examples to help the reader understand the ALCR mechanism and its associated costs. Section 5.2 describes the proposed mathematical model and the numerical approach. Specifically, the optimum checkpoint interval is discussed in Section 5.2.3. Section 5.3 presents numerical examples and Section 5.4 presents conclusions and future research.

5.1 Indicative Examples

In this section, some examples are provided regarding the changes that should be performed to the source code of a software application, in order to add checkpoints into long-running loops, using actual ALCR libraries. Their purpose is to help the reader understand the overall concept of the ALCR mechanism, and also to explain why the arbitrary selection of the checkpoint interval may negatively affect the execution time of a software application. These examples are also expected to facilitate the understanding of the proposed mathematical model that is presented in Section 5.2, as well as the numerical examples that are provided in Section 5.3.

Figure 5.1 demonstrates how the source code of a software application should be modified, in order to insert an application-level checkpoint in a long-running loop. As can be seen by the given example, a non-trivial number of library-specific statements (marked with red color) should be added to the program, in order to insert a checkpoint in the selected loop. The example is based on a specific ALCR library called CRAFT [33]. However, similar modifications are also required by other ALCR libraries, as can be seen by Figure 5.2.

The examples in Figures 5.1 and 5.2 show that the insertion of checkpoints into long-running loops necessitates significant source code modifications. In fact, a number of statements should be inserted, in order to determine: (i) the location of the checkpoints, (ii) what data should be safely stored every time a checkpoint is generated, and (iii) how frequently a checkpoint should

<pre> #include <mpi.h> int main(int argc, char* argv[]) { int n=5, iteration=1; double dbl = 0.0; int * dataArr = new int[n]; for (; iteration <= 100; iteration++) { // Computation-communication loop modifyData(&dbl, dataArr); } return EXIT_SUCCESS; } </pre>	<pre> #include <mpi.h> #include <craft.h> int main(int argc, char* argv[]) { int n=5, iteration=1, cpFreq=10; double dbl = 0.0; int * dataArr = new int[n]; // ===== DEFINE CHECKPOINT ===== // Checkpoint myCP("myCP", MPI_COMM_WORLD); myCP.add("dbl", &dbl); myCP.add("iteration", &iteration); myCP.add("dataArr", &dataArr); myCP.commit(); myCP.restartIfNeeded(&iteration); for (; iteration <= 100; iteration++) { // Computation-communication loop modifyData(&dbl, dataArr); myCP.updateAndWrite(iteration, cpFreq); } return EXIT_SUCCESS; } </pre>
---	--

Figure 5.1: The additional code that should be inserted for adding an application-level checkpoint in a lengthy loop, using CRAFT (Adapted from [33]).

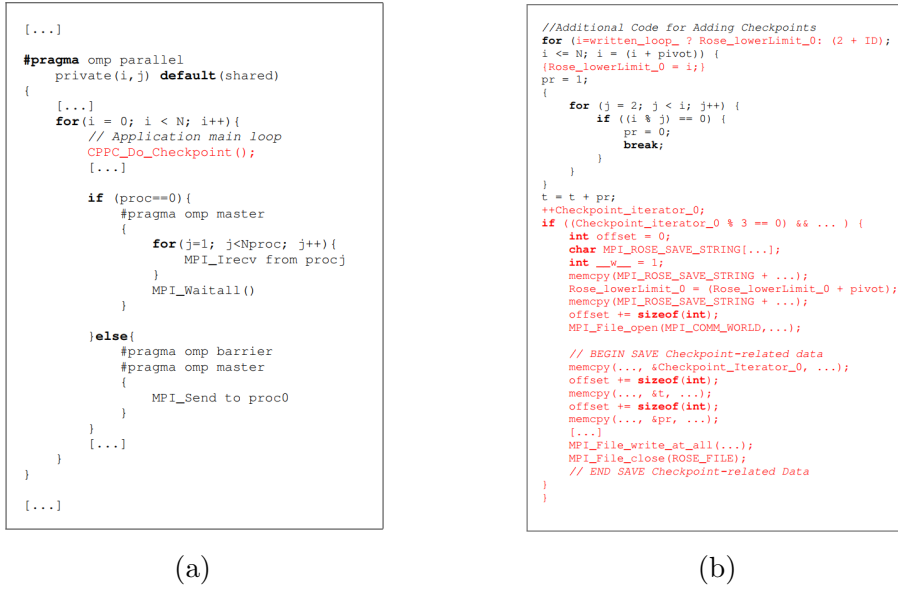


Figure 5.2: The additional source code (marked with red color) required for inserting an application-level checkpoint into a loop, using: (a) an extension of the CPPC tool proposed by Losada et al. [122], and (b) the ITALC ALCR library [34].

be generated (e.g., the value of `cpFreq` in Figure 5.1). These additional statements are *methods* of a specific library, and may require several operations with significant execution time. For instance, the `updateAndWrite()` method in Figure 5.1, is expensive in execution time, since the creation of a checkpoint usually requires multiple memory accesses [33,34]. Thus the execution time of checkpointing must be added to the overall execution time of the software application. If failures are very infrequent and the cost of checkpointing is high, frequent checkpoints will result in an average execution time of the application that is higher than the same application which runs without checkpoints. Hence, the checkpoint interval should be optimally selected, in order to avoid the introduction of execution time overhead. The total execution time of a program which contains checkpoints in its loops is presented in Section 5.2, and numerical examples are provided in Section 5.3.

5.2 Expected Execution Time of a Program Without and With Checkpoints

Consider a program P that executes a total of M instructions; it may contain loops so that M is the total number of instructions it executes. Assume that when the execution starts,

there is an overhead associated with loading its data and code into memory, which consumes A time units. If the program is executed without any errors or failures, and if each instruction is executed in c time units, then the total execution time for P will be:

$$T(P) = A + cM. \quad (5.1)$$

Now suppose that no failures or errors occur during the initial and final durations A , B , however with probability g there may be a failure in any one of the instructions. We assume that the failure is detected after a delay which takes δ time units.

5.2.1 Expected Execution Time Without Checkpoints

When a failure is detected, the program has to be re-executed, and if the failures occur during further executions, the execution may have to be repeated several times. Let $\tau(P)$ denote the total execution time of the program, and let $E\tau(P)$ be its expected value. Then:

$$\begin{aligned} E\tau(P) &= (1 - g)^M(A + cM + \delta) \\ &+ \sum_{u=1}^M (1 - g)^{u-1} g[A + c.u + \delta + E\tau(P)], \end{aligned} \quad (5.2)$$

$$\begin{aligned} &= A + (1 - g)^M c.M + \delta + [1 - (1 - g)^M] E\tau(P) \\ &+ c \left[\frac{1 - (1 - g)^M}{g} - M(1 - g)^M \right], \text{ hence} \\ E\tau(P) &= \frac{A + \delta}{(1 - g)^M} + c \cdot \frac{1 - (1 - g)^M}{g(1 - g)^M}. \end{aligned} \quad (5.3)$$

If a failure occurs, this only becomes known after δ time units, the program has to be restarted and run again, so that the time $A + c.u + \delta$ has been wasted. When there are no failures we see from (5.3) that

$$E\tau(P) = A + \delta + M, \quad (5.4)$$

since:

$$\lim_{g \rightarrow 0} \frac{1 - (1 - g)^M}{g(1 - g)^M} = M. \quad (5.5)$$

When g is very small so that $gM \ll 1$, we can use the following approximation directly from (5.3):

$$E\tau(P) \approx \frac{A + \delta + c.M}{1 - g.M}. \quad (5.6)$$

5.2.2 Estimating the Failure Probability g

In order to use the above expressions, we will need $(1 - g)^M$ the probability that no error or failure will occur during the program's execution, and the probability that at least one failure occurs during the program's execution is $F = 1 - (1 - g)^M$. Note that the notion of a failure in this case is that of any event that stops the execution of the program and which arises from the program's execution environment, i.e., the platform. If $gM \ll 1$ then $F \approx gM$.

The value of g can be estimated as follows. Take a simple linear code that executes M instructions, and then repeats the execution, i.e. a single loop containing M sequential instructions. This code should not contain any ALCR or other checkpointing constructs.

1. Run the program repeatedly. Each time the program returns to the first instruction, increment the counter $N \leftarrow N + 1$.
2. If the program execution stops, increment a counter $N_F \leftarrow N_F + 1$. Update $g \approx \frac{N_F}{N.M}$.
3. Then restart the program at its initial instruction and set $N \leftarrow N + 1$.

5.2.3 Optimum Checkpoints

When the program must run for a long time, i.e. when M is large of failure Mg cannot be neglected, checkpoints can be placed at periodic intervals, say after K instructions are executed, but they result in a cost $B(K)$ in the amount of time needed to create the checkpoint, since the status of the program and all its data must be saved. $B(K)$ may be an increasing function of K when the data that the program has modified during the interval of execution of K instructions needs to be saved. Thus the program will now execute a total of M instructions in successive blocks of $b(M, K) = \lceil \frac{M}{K} \rceil$ instructions, all of which are of length K , except for the last one of length $K_o = M - K[\lceil \frac{M}{K} \rceil - 1]$.

Applying the previous analysis, we compute the total average execution time of the program with checkpoints:

$$\begin{aligned}
 E_{cp}\tau(P) &= \frac{A + \delta}{(1 - g)^K} + c \frac{1 - (1 - g)^K}{g(1 - g)^K} \\
 &+ [b(M, K) - 2] \left[\frac{B(K) + \delta}{(1 - g)^K} + c \frac{1 - (1 - g)^K}{g(1 - g)^K} \right] \\
 &+ \frac{B(K) + \delta}{(1 - g)^{K_o}} + c \frac{1 - (1 - g)^{K_o}}{g(1 - g)^{K_o}}, \\
 &= \frac{A + \delta}{(1 - g)^K} + \frac{[b(M, K) - 2][B(K) + \delta]}{(1 - g)^K} \\
 &+ c \frac{[b(M, K) - 1][1 - (1 - g)^K]}{g(1 - g)^K} \\
 &+ \frac{B(K) + \delta}{(1 - g)^{K_o}} + c \frac{1 - (1 - g)^{K_o}}{g(1 - g)^{K_o}}.
 \end{aligned} \tag{5.7}$$

Therefore optimum checkpoint interval K^* is the value of K that minimizes $E_{cp}\tau(P)$, which can be computed numerically from equation (5.7).

In order to better illustrate the benefit of the ALCR we also define the percentage *Gain*:

$$Gain = \frac{E\tau(P) - E_{cp}\tau(P)}{E\tau(P)} \times 100, \tag{5.8}$$

where $E\tau(P)$ is the expected execution time of the program (or software application) P when ALCR is not used.

5.2.4 Program with a Long Loop

Suppose that a program contains a single loop with L instructions that is executed repeatedly n times so that the program executes $M = n.L$ instructions. If a checkpoint is inserted for each I loops so that the block of executed instructions between checkpoints is of length $K = I.L$, then a total of $b(nL, IL) = \lceil \frac{n}{I} \rceil - 1$ checkpoints are placed, since the start of the loop will in itself require a checkpoint. From equation (5.9) with $M = n.L$ and $K = I.L$ we have:

$$\begin{aligned}
 E_{cp}\tau(P) &= \frac{[b(M, K) - 1][B(K) + \delta]}{(1 - g)^K} \\
 &+ c \frac{[b(M, K) - 1][1 - (1 - g)^K]}{g(1 - g)^K} + \frac{B(K) + \delta}{(1 - g)^{K_o}} \\
 &+ c \frac{1 - (1 - g)^{K_o}}{g(1 - g)^{K_o}}.
 \end{aligned} \tag{5.9}$$

If the number of instructions that are executed during a single loop iteration is L , the optimum number of iterations between two successive checkpoints is $I^* = \frac{K^*}{L}$.

5.2.5 Nested Loops

Suppose that we identify, either manually or using an ALCR library, that the best location for adding checkpoints is a loop that contains one or more internal loops. These internal loops can be treated in a black-box manner as normal statements (e.g., method calls), which require the execution of a number of instructions. The number of instructions executed in the internal loops can be used to calculate the values of L and M of the selected outer loop yielding the optimum number of loop iterations between checkpoints I^* .

Another possible approach, especially in the case of nested loops with multiple levels of nesting, is to treat the nested loops individually. In this case, checkpoints could be generated in the innermost loop. However, the latter approach would require additional development effort, in order to map iterations to actual instructions, since ALCR libraries usually focus on iterations.

5.3 Numerical Examples

In this section, a set of numerical examples illustrate the effect of the checkpoint interval K on the expected execution time of a software application. In Figure 5.3, the case of a software application with a relatively small loop having $M = 1000$ is presented. Figure 5.3a compares the expected execution time of the application with and without the ALCR mechanism for different values of K , while Figure 5.3b shows the expected *Gain* of Section 5.2.3 for different values of K . The values that correspond to the optimum checkpoint interval K^* are marked within a rectangle. Figure 5.3 illustrates the fact that the optimum checkpoint interval K^* minimizes the overall execution time of the application and maximizes the overall expected *Gain*. Therefore, the ALCR mechanism will not reduce the expected execution time of a given software application unless the checkpoint interval is optimally selected. Indeed, for some poorly chosen values of K , the expected execution time of the application with checkpointing is higher than the expected execution time of the same application without checkpoints. Similar observations can be made for software with longer loops in Figures 5.4, 5.5 and 5.6. This emphasizes the importance of setting K to be close or at K^* .

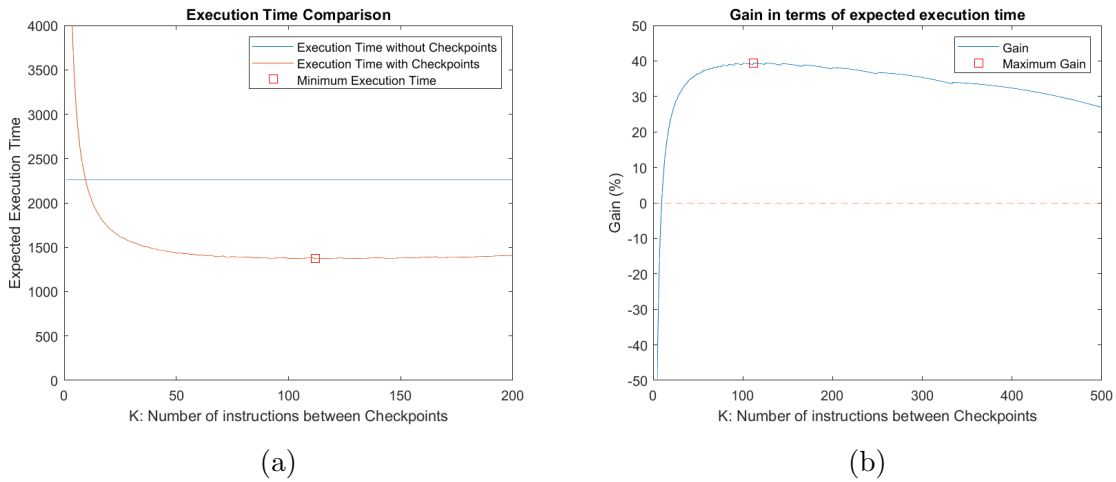


Figure 5.3: Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K , for a software application having a small computational loop with $M = 1000$.

The examples of Figures 5.3, 5.4, 5.5 and 5.6 show that a significant reduction in the execution time of a software application can be achieved by the ALCR mechanism, if the checkpoint interval is selected to be at, or close to, the optimum K^* . In these examples the Gain, ranges

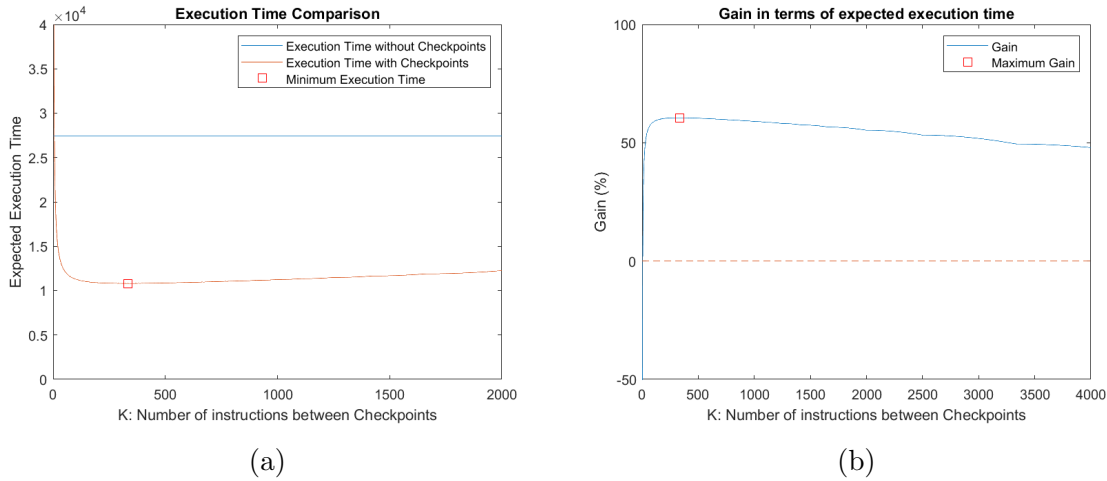


Figure 5.4: Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K , for a software application having a medium length computational loop with $M = 10^4$.

from 40% to 60%. However, suboptimal values of the checkpoint interval will lead to a smaller Gain or even to an average execution time which is larger than when ALCR is not used. Indeed, the checkpoint interval should not be selected arbitrarily and must be tuned to a value at, or close to, the optimum K^* .

5.4 Summary and Critical Analysis

This section proposed a method for setting the checkpoint intervals in ALCR for software applications which contain long-running loops, and which run on platforms that are subject to failures. We have shown that the optimum checkpoint interval which minimizes the expected execution time of the program, depends on various parameters which can be incorporated into a single numerical expression. The expression can then be used as part of an ALCR tool to compute the optimum checkpoint interval for each individual loop in the program. The approach can be used through a set of MATLAB scripts that calculate the optimum checkpoint interval of different computational loops. We have illustrated our results via different numerical examples.

Several directions for future work can be considered. Energy consumption is a critical property that will be affected by checkpointing, as well as by the re-execution of a program in case of

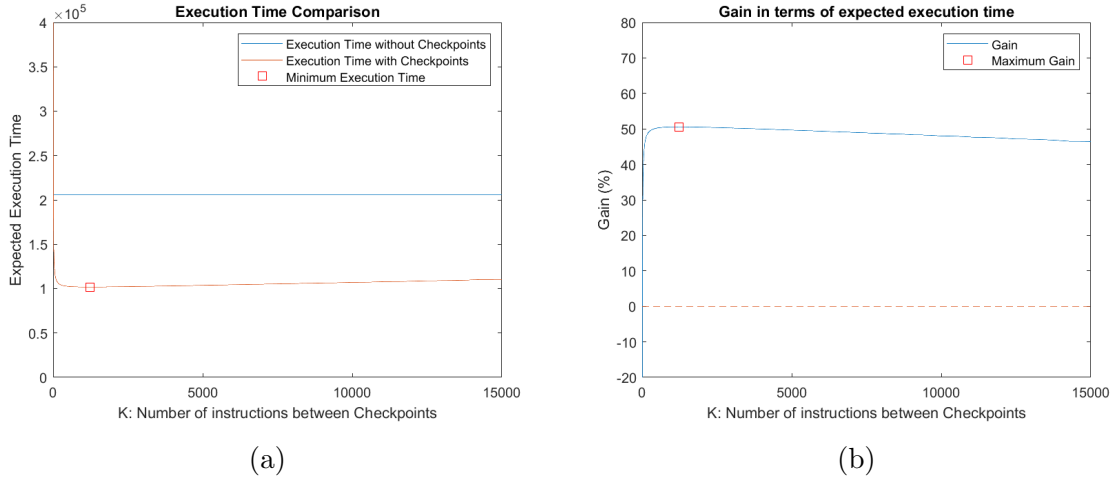


Figure 5.5: Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K , for a software application with a large computational loop with $M = 10^5$.

failure. Thus in recent work [180], checkpointing was considered from the perspective of its effect on energy consumption. Further research is needed to see how the checkpoint interval can be selected to achieve a compromise between energy consumption and execution times. The issue can become quite intricate if we consider the effect of the secondary memory medium. Since checkpointing will generally increase the use of secondary memory, and secondary memory related failures may increase with the amount of usage, a platform where many applications use ALCR, may have a failure rate which increases with usage and age. Thus a time dependent value of the failure probability g will need to be considered in this case. Furthermore, with rotating secondary memories and some other devices, the use of ALCR will also increase energy consumption. These are all questions that require further research.

Another interesting area of investigation is the use of ALCR to restart applications after attacks. Indeed, ALCR could perhaps be used to disrupt the attacker (e.g. [181]), but in turn attackers may exploit checkpointing (e.g. [182]) to create an increase in workload in a system, leading to a form of Denial of Service through workload saturation. Thus the interaction of ALCR and checkpointing in general, and security, is also a worthwhile subject of investigation.

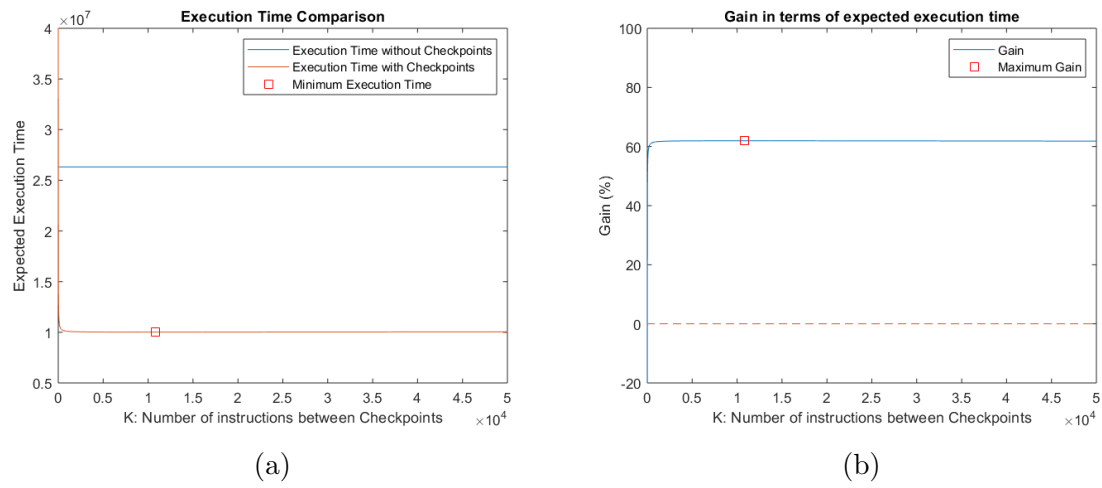


Figure 5.6: Comparison of the expected execution time (a), and the Gain in the expected execution time (b), plotted versus the value of K , for a software application with a very large computational loop with $M = 10^7$.

Chapter 6

Conclusion

The purpose of the present thesis was to advance the state of the art in the fields of software security and reliability, by investigating novel approaches that facilitate the development of secure and reliable software, as well as by addressing existing open issues. More specifically, as far as software security is concerned, the present thesis focused on the development of a model that allows the quantitative evaluation of the internal security level of software applications. It also investigated the ability of software-related factors (including previously uninvestigated factors) to indicate the existence of specific types of software vulnerabilities, whereas it further examined the discriminative power of previously studied vulnerability indicators based on new datasets and approaches. Regarding software reliability, this thesis highlighted the impact that the arbitrary selection of the checkpoint interval may have on the performance (i.e., execution time) of software programs that adopt the ALCR mechanism and proposed a mathematical model for the calculation of the optimum checkpoint interval. In the following, Section 6.1 provides a summary of the present thesis and a detailed discussion of its main research outcomes, whereas Section 6.2 presents some interesting directions for future work.

6.1 Thesis Summary and Achievements

In Chapter 2, we provided a comprehensive literature review of the three main concepts (i.e., subfields) that are relevant to the broader topic of the present research, namely: (i) the quantitative security assessment, (ii) the vulnerability prediction, and (iii) the optimum checkpoint

recommendation. The most representative contributions in each one of these subfields were described and critically analyzed, while their most important open issues were identified and reported. Special emphasis was given on the open issues that the present research attempts to address.

In Chapter 3, an empirical study was performed with the main purpose (i) to investigate the ability of software metrics to indicate the existence of specific types of vulnerabilities in software products, and (ii) to examine whether potential interdependencies exist between different vulnerability types. Another goal was to evaluate on a much larger code base the ability of software metrics to highlight software vulnerabilities in general. For this purpose, a large benchmark repository comprising 100 popular Java applications was constructed and statically analyzed using popular open source tools in order to calculate a set of software metrics (including previously uninvestigated metrics) and the densities of a set of carefully constructed weakness categories. Both security- and non-security-specific weakness categories were considered, in order to lead to more reliable observations. Correlation analysis was employed in order to investigate whether statistically significant positive correlations exist between the software metrics and the defined weakness categories, as well as between the weakness categories themselves.

The results of the aforementioned empirical study indicate that software metrics can be used only as weak indicators of specific vulnerability types, while important interdependencies may exist between different vulnerability types. They also suggest that software metrics are capable of discriminating between security-specific and non-security-specific weaknesses, and thus they may be sufficient indicators of vulnerabilities in general, while previously uninvestigated metrics may also be good vulnerability indicators. The last two observations are important for the rest of the present research, as they denote that software metrics are valuable indicators of software security, and therefore they could possibly be used for enhancing the completeness of security assessment models. To the best of our knowledge, this is the first study in the field of vulnerability prediction that focuses on the relationships between specific vulnerability types, while it is the largest study in terms of code base size.

In Chapter 4, a hierarchical Security Assessment Model (SAM) was proposed, which is able to assess the internal security of software products written in Java programming language, based on low-level indicators (i.e., static analysis alerts and software metrics). The model, following the guidelines of ISO/IEC 25010 [26], and based on a set of thresholds and weights,

systematically aggregates these low-level indicators, to produce a high-level security score (i.e., the *Security Index*) that reflects the internal security of the analyzed software. The model is practical, since it is fully-automated and operationalized in the form of standalone tools [38], and as part of a broader CASE platform. In order to enhance its reliability, the model thresholds were calibrated based on the benchmark repository of 100 Java applications produced for the purposes of the empirical study that was described in Chapter 3. In addition, its weights were elicited in a way to chiefly reflect the knowledge expressed by CWE [41] (instead of expert judgments), through a novel weight elicitation approach grounded on the AHP [162] and SMARTS/SMARTER [166] decision-making techniques.

The proposed SAM was evaluated through a set of experiments conducted on a large repository of 150 open-source software applications retrieved from GitHub. The results of the analysis revealed the capacity of the proposed model to reliably assess the internal security of software products, with sufficient discretion power. In addition, the ability of the proposed model to indicate the existence of vulnerabilities in software components (i.e., classes), and therefore provide reliable class-level security estimations was also studied. For this purpose, a highly balanced dataset comprising 600 vulnerable and 600 clean classes was constructed based on the OWASP Benchmark [121], and was analyzed using the proposed SAM. Discriminant analysis was applied which revealed the ability of the proposed model to discriminate between vulnerable and clean classes. Hence, the proposed SAM can potentially be used as the basis for the conduction of transparent (i.e., non-ML-based) vulnerability prediction. It should be noted that since the model is based on static analysis alerts and software metrics, the latter also provides further support to the findings of previous studies (e.g. [56, 66]) regarding the ability of these factors (and especially their combination) to be used as vulnerability indicators.

To the best of our knowledge this is the first fully-automated and operationalized security assessment model that can be found in the related literature, while it is the first security evaluation approach, in general, that was built and evaluated on such a large volume of empirical data (i.e., 250 open-source software applications comprising approximately 20 million lines of code). The detailed results of the analyzed software products, as well as the produced vulnerability dataset, have been made publicly available in order to facilitate further research in the field [38].

In Chapter 5, we investigated the impact that the adoption of the Application-level Checkpoint

and Restart (ALCR) mechanism, and particularly the selection of the inter-checkpoint interval may have on the performance (i.e., execution time) of a software application. Emphasis was given on the case of long-running loops, since existing ALCR libraries suggest the insertion of checkpoints in these locations, as computational loops constitute the main source of failure related re-executions. Initially, a set of real-world examples were provided in order to help the reader understand what code modifications are required for adding application-level checkpoints in long-running loops using actual ALCR libraries, and, in turn, to clarify why suboptimal values of the checkpoint interval may lead to execution time overheads.

A mathematical model was then proposed for calculating the optimum checkpoint interval, i.e., the interval between two successive checkpoints that minimizes the expected execution time of programs with loops. More specifically, we have shown that the optimum checkpoint interval depends on various parameters which can be incorporated into a single numerical expression. These parameters include: (i) the program failure rate, (ii) the execution cost for establishing a checkpoint, and (iii) the execution cost for restarting a program after a failure. The produced expression can be used to compute the optimum checkpoint interval for each individual loop in the program. A set of MATLAB scripts have been implemented for facilitating the utilization of the proposed model [38].

Based on the proposed mathematical model, a number of numerical examples were provided with the purpose to illustrate the effect of the checkpoint interval on the expected execution time of a software application. The numerical examples suggest that the checkpoint interval should be selected to be at, or close to, its optimum value in order to achieve a significant reduction in the expected execution time of the application. They also suggest that the arbitrary selection of the checkpoint interval should be avoided, since suboptimal values of the checkpoint interval may even lead to an expected execution time higher than the expected execution time of the same application that does not adopt checkpointing. To the best of our knowledge, this is the first study that formally assesses the impact of ALCR on the performance (i.e., execution time) of software applications, and that proposes a method for the calculation of the optimum checkpoint interval of application-level checkpoints.

6.2 Future Work

Based on the outcomes of the present thesis, several directions for future work can be identified. These directions are described in the following:

- The empirical study presented in Chapter 3 was based on software applications written in Java programming language. Hence, an interesting direction for future work would be to replicate the present study by using software applications written in programming languages other than Java, in order to investigate the generalizability of its results. In addition, the main observation of the study was that important interdependencies may exist among different vulnerability types. Knowing these interdependencies may be valuable for the production of more secure software since closely related security issues (i.e., potential vulnerabilities) may be mitigated together. To this end, as a future research attempt, we plan to create a framework able to analyze a given code base and inform developers and project managers for underlying statistically significant interdependencies between specific vulnerability types.
- The Security Assessment Model (SAM) presented in Chapter 4 is able to assess the internal security level of software applications written in Java programming language. Thus, future work will focus on extending the proposed model in order to support the evaluation of applications written in other programming languages. Specific emphasis will be given on C/C++, which are known unsafe languages [183–185] that are widely used by modern embedded and IoT systems. Furthermore, another interesting direction for future work is to investigate how the proposed model that was designed for internal security assessment can be combined with the Attack Surface Metric (ASM), which is a commonly accepted external security evaluation indicator. This would allow the construction of a broad security assessment framework that would enable the holistic evaluation of the overall security level of software products (both internal and external). Finally, in Section 4.3.4, the ability of the proposed SAM to discriminate between vulnerable and clean software components was highlighted. Hence, an interesting topic would be to investigate whether the proposed model can be used for the construction of a transparent VPM. This can potentially be achieved, for example, by training a specific threshold for the produced

security score (i.e., *Security Index*), which can be then used for the classification of the components as vulnerable or clean.

- The identification of novel indicators of software security is another interesting research topic. In particular, previously uninvestigated software-related factors (e.g., Technical Debt [186] and popularity) could be studied regarding their relationship to software security. The proposed SAM can potentially be used as the reference point of these studies with respect to software security.
- In Chapter 5, a model for calculating the checkpoint interval that optimizes the performance (i.e., average execution time) of applications that adopt the ALCR mechanism was proposed. However, apart from performance, energy consumption is another critical property that will be affected by checkpointing, as well as by the re-execution of a program in case of failure. Therefore, further research will be conducted to see how the checkpoint interval can be selected to achieve a compromise between energy consumption and execution times. In addition, in this study the probability of failure g was assumed to be constant. However, since the failure rate is expected to increase with time, mainly due to failures caused by the increased usage of the secondary memory, an interesting direction for future work would be to consider a time-dependent failure probability g . Finally, since checkpointing, and particularly ALCR, can be used both to facilitate and to disrupt attacks, the interaction of the ALCR and security is another worthwhile topic of investigation.

Bibliography

- [1] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.
- [2] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, Jan 2004.
- [3] G. McGraw, “On bricks and walls: Why building secure software is hard,” *Computers & Security*, vol. 21, no. 3, pp. 229–238, 2002.
- [4] P. Salini and S. Kanmani, “Survey and analysis on security requirements engineering,” *Computers and Electrical Engineering*, vol. 38, no. 6, pp. 1785–1797, 2012.
- [5] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, “Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study,” *Proceedings of the 22Nd ACM SIG-SOFT International Symposium on Foundations of Software Engineering*, pp. 257–268, 2014.
- [6] G. J. Holzmann, “The Value of Doubt,” *IEEE Software*, vol. 34, no. 1, pp. 106–109, 2017.
- [7] P. Anderson, “Measuring the value of static-analysis tool deployments,” *IEEE Security and Privacy*, vol. 10, no. 3, pp. 40–47, 2012.
- [8] I. Krsul, *Software Vulnerability Analysis*. PhD thesis, Department of Computer Sciences, Purdue University, 1998.
- [9] B. Chess and G. McGraw, “Static analysis for security,” *Security & Privacy, IEEE*, vol. 2, pp. 76–79, 2004.
- [10] M. Howard, D. LeBlanc, and J. Viega, *24 Deadly Sins of Software Security*. McGraw-Hill, 2010.

- [11] B. Boehm and V. R. Basili, “Software Defect Reduction Top 10 List,” *Computer*, vol. 34, pp. 135–137, 2001.
- [12] G. Wurster and P. C. van Oorschot, “The developer is the enemy,” *NSPW ’08: Proceedings of the 2008 Workshop on New Security Paradigms*, pp. 89–97, 2008.
- [13] M. Green and M. Smith, “Developers are Not the Enemy!: The Need for Usable Security APIs,” *IEEE Security and Privacy*, vol. 14, 2016.
- [14] G. McGraw, “Software security,” *Datenschutz und Datensicherheit - DuD*, vol. 36, pp. 662–665, Sep 2012.
- [15] G. McGraw, “Automated code review tools for security,” *Computer*, vol. 41, no. 12, 2008.
- [16] M. Howard, *Writing secure code*. Redmond, Wash: Microsoft Press, 2003.
- [17] M. Howard and S. Lipner, *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press, 2006.
- [18] B. Arkin, S. Stender, and G. McGraw, “Software penetration testing,” *IEEE Security and Privacy*, vol. 3, no. 1, pp. 84–87, 2005.
- [19] B. P. Miller, L. Fredriksen, and B. So, “An Empirical Study of the Reliability of UNIX Utilities,” *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.
- [20] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, “Security Testing: A Survey,” *Advances in Computers*, vol. 101, pp. 1–51, 2016.
- [21] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?,” *Proceedings of the 2013 International Conference on Software Engineering*, pp. 672–681, 2013.
- [22] R. Bholanath, “Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software,” *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 470–481, 2016.
- [23] P. Morrison, D. Moye, R. Pandita, and L. Williams, “Mapping the field of software life cycle security metrics,” *Information and Software Technology*, vol. 102, no. May, pp. 146–159, 2018.

- [24] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner, “Software quality models: Purposes, usage scenarios and requirements,” in *Proceedings - International Conference on Software Engineering*, pp. 9–14, 2009.
- [25] H. Xu, J. Heijmans, and J. Visser, “A practical model for rating software security,” *Proceedings - 7th International Conference on Software Security and Reliability Companion, SERE-C 2013*, pp. 231–232, 2013.
- [26] ISO/IEC, “ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models,” tech. rep., 2011.
- [27] M. Jimenez, M. Papadakis, and Y. L. Traon, “Vulnerability Prediction Models : A case study on the Linux Kernel,” in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 1–10, 2016.
- [28] B. Randell, “System Structure for Software Fault Tolerance,” *Science*, no. 2, pp. 1–18, 1975.
- [29] L. Chen and A. Avizienis, “N-version programming: A fault-tolerance approach to reliability of software operation,” in *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, vol. 1, pp. 3–9, 1978.
- [30] A. Avizienis, “Fault-tolerance and Fault-intolerance: Complementary Approaches to Reliable Computing,” *SIGPLAN Not.*, vol. 10, no. 6, pp. 458–464, 1975.
- [31] I. P. Egwuotuoha, D. Levy, B. Selic, and S. Chen, “A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems,” *Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.
- [32] J. Walters and V. Chaudhary, “Application-Level Checkpointing Techniques for Parallel Programs,” *Distributed Computing and Internet Technology*, pp. 221–234, 2006.
- [33] F. Shahzad, J. Thies, and G. Wellein, “CRAFT: A library for easier application-level Checkpoint/Restart and Automatic Fault Tolerance,” *IEEE Transactions on Parallel and Distributed Systems*, 2018.

- [34] R. Arora and T. N. Ba, “ITALC : Interactive Tool for Application - Level Checkpointing,” *Proceedings of the Fourth International Workshop on HPC User Support Tools*, 2017.
- [35] J. W. Young, “A First Order Approximation to the Optimum Checkpoint Interval,” *Commun. ACM*, vol. 17, no. 9, pp. 530–531, 1974.
- [36] E. Gelenbe, “On the Optimum Checkpoint Interval,” *Journal of the ACM*, vol. 26, no. 2, pp. 259–270, 1979.
- [37] E. Gelenbe and M. Hernandez, “Optimum Checkpoints With Age-Dependent Failures,” *Acta Informatica*, vol. 531, pp. 519–531, 1990.
- [38] “Supporting Material.” [Online]. Available: <https://sites.google.com/view/phd-ms-icl>.
- [39] M. Siavvas, M. Jankovic, D. Kehagias, and D. Tzovaras, “Is Popularity an Indicator of Software Security?,” in *2018 IEEE 9th International Conference on Intelligent Systems (IS)*, 2018.
- [40] M. Siavvas, D. Tsoukalas, M. Jankovic, D. Kehagias, A. Chatzigeorgiou, D. Tzovaras, N. Anicic, and E. Gelenbe, “An Empirical Evaluation of the Relationship between Technical Debt and Software Security,” *9th International Conference on Information Society and Technology (ICIST) 2019*, 2019.
- [41] “Common Weakness Enumeration (CWE).” [Online]. Available: <https://cwe.mitre.org/>.
- [42] M. Siavvas, E. Gelenbe, D. Kehagias, and D. Tzovaras, “Static analysis-based approaches for secure software development,” in *Security in Computer and Information Sciences*, (Cham), pp. 142–157, Springer International Publishing, 2018.
- [43] T. DeMarco, *Controlling Software Projects: Management, Measurement, and Estimates*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1986.
- [44] P. K. Manadhata and J. M. Wing, “An attack surface metric,” *IEEE Transactions on Software Engineering*, vol. 37, no. 3, 2011.
- [45] M. Howard, J. Pincus, and J. M. Wing, “Measuring relative attack surfaces,” in *Computer Security in the 21st Century*, pp. 109–137, 2005.

- [46] P. K. Manadhata and J. M. Wing, “Measuring a Systems Attack Surface,” tech. rep., 2004.
- [47] C. Theisen, N. Munaiah, M. Al-Zyoud, J. C. Carver, A. Meneely, and L. Williams, “Attack surface definitions: A systematic literature review,” *Information and Software Technology*, vol. 104, pp. 94 – 103, 2018.
- [48] G. Hatzivasilis, I. Papaefstathiou, and C. Manifavas, “Software security, privacy, and dependability: Metrics and measurement,” *IEEE Software*, vol. 33, pp. 46–54, July 2016.
- [49] N. Munaiah and A. Meneely, “Beyond the Attack Surface: Assessing Security Risk with Random Walks on Call Graphs,” *Proceedings of the 2016 ACM Workshop on Software PROtection*, pp. 3–14, 2016.
- [50] Y. Shin and L. Williams, “Is complexity really the enemy of software security?,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2008.
- [51] Y. Shin and L. Williams, “An empirical model to predict security vulnerabilities using code complexity metrics,” in *ESEM’08: Proceedings of the 2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2008.
- [52] V. H. Nguyen and L. M. S. Tran, “Predicting vulnerable software components with dependency graphs,” *Proceedings of the 6th International Workshop on Security Measurements and Metrics - MetriSec ’10*, 2010.
- [53] Y. Shin and L. Williams, “An initial study on the use of execution complexity metrics as indicators of software vulnerabilities,” in *Proceedings - International Conference on Software Engineering*, 2011.
- [54] I. Chowdhury and M. Zulkernine, “Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?,” in *Proc. of the 2010 ACM Symposium on Applied Comp.*, 2010.
- [55] I. Chowdhury and M. Zulkernine, “Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities,” *Journal of Systems Architecture*, vol. 57, pp. 294–313, 2011.

- [56] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.
- [57] S. Moshtari, A. Sami, and M. Azimi, "Using complexity metrics to improve software security," *Computer Fraud and Security*, 2013.
- [58] S. Moshtari and A. Sami, "Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction," *Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC '16*, pp. 1415–1421, 2016.
- [59] H. Alves, B. Fonseca, and N. Antunes, "Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study," *Proceedings - 2016 12th European Dependable Computing Conference, EDCC 2016*, 2016.
- [60] I. Chowdhury, B. Chan, and M. Zulkernine, "Security metrics for source code structures," *Proceedings of the fourth international workshop on Software engineering for secure systems - SESS '08*, pp. 57–64, 2008.
- [61] B. Alshammari, C. Fidge, and D. Corney, "Security metrics for object-oriented class designs," in *2009 Ninth International Conference on Quality Software*, pp. 11–20, Aug 2009.
- [62] B. Alshammari, C. Fidge, and D. Corney, "Security metrics for object-oriented designs," in *2010 21st Australian Software Engineering Conference*, pp. 55–64, April 2010.
- [63] A. A. Abdulrazeg, N. M. Norwawi, and N. Basir, "Security metrics to improve misuse case model," *Proceedings 2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic, CyberSec 2012*, pp. 94–99, 2012.
- [64] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers and Security*, vol. 26, no. 3, pp. 219–228, 2007.
- [65] J. Walden, M. Doyle, G. A. Welch, and M. Whelan, "Security of open source web applications," *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009.

- [66] M. Gegick, P. Rotella, and L. Williams, “Predicting attack-prone components,” *Proceedings - 2nd International Conference on Software Testing, Verification, and Validation, ICST 2009*, pp. 181–190, 2009.
- [67] Y. Tang, F. Zhao, Y. Yang, H. Lu, Y. Zhou, and B. Xu, “Predicting Vulnerable Components via Text Mining or Software Metrics? An Effort-Aware Perspective,” *Proceedings - 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015*, pp. 27–36, 2015.
- [68] J. Walden and M. Doyle, “SAVI: Static-Analysis Vulnerability Indicator,” *IEEE Security Privacy*, vol. 10, pp. 32–39, May 2012.
- [69] “Fortify SCA.” [Online]. Available: <https://software.microfocus.com/en-us/software/sca>. Last Accessed: 01-03-2019.
- [70] P. Mell, K. Scarfone, and S. Romanosky, “Common Vulnerability Scoring System,” *IEEE Security & Privacy Magazine*, vol. 4, no. 6, pp. 85–89, 2006.
- [71] P. Mell, K. Scarfone, and S. Romanosky, “A Complete Guide to the Common Vulnerability Scoring System Version 2.0,” *FIRSTForum of Incident Response and Security Teams*, pp. 1–23, 2007.
- [72] V. Verendel, “Quantified security is a weak hypothesis,” *Proceedings of the 2009 workshop on New security paradigms workshop - NSPW '09*, p. 37, 2009.
- [73] N. Munaiah, F. Camilo, W. Wigham, A. Meneely, and M. Nagappan, “Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project,” *Empirical Software Engineering*, vol. 22, no. 3, 2017.
- [74] S. Wagner, *Software product quality control*. Heidelberg: Springer, 2013.
- [75] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merrit, *Characteristics of Software Quality*. Amsterdam, Netherlands: North-Holland, 1976.
- [76] J. A. McCall, P. K. Richards, and G. F. Walters, *Factors in Software Quality*. Springfield: National Technical Information Service, 1977.
- [77] R. G. Dromey, “A model for software product quality,” *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 146–162, 1995.

- [78] J. Bansiya and C. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.
- [79] I. Heitlager, T. Kuipers, and J. Visser, "A Practical Model for Measuring Maintainability," *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pp. 30–39, 2007.
- [80] S. Wagner, K. Lochmann, L. Heinemann, M. Klas, A. Trendowicz, R. Plosch, A. Seidi, A. Goeb, and J. Streit, "The Quamoco product quality modelling and assessment approach," *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1133–1142, 2012.
- [81] S. Wagner, A. Goeb, L. Heinemann, M. Kläs, C. Lampasona, K. Lochmann, A. Mayr, R. Plösch, A. Seidl, J. Streit, and A. Trendowicz, "Operationalised product quality models and assessment: The Quamoco approach," *Information and Software Technology*, vol. 62, pp. 101–123, 2015.
- [82] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *IEEE International Conference on Software Maintenance, ICSM*, 2010.
- [83] M. Siavvas, K. Chatzidimitriou, and A. Symeonidis, "QATCH - An adaptive framework for software product quality assessment," *Expert Systems with Applications*, vol. 86, 2017.
- [84] S. Wagner, "A Bayesian network approach to assess and predict software quality using activity-based quality models," *Proceedings of the 5th International Conference on Predictor Models in Software Engineering - PROMISE '09*, p. 1, 2009.
- [85] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, pp. 287–307, Jun 2012.
- [86] C. Jin and S.-W. Jin, "Software reliability prediction model based on support vector regression with improved estimation of distribution algorithms," *Applied Soft Computing*, vol. 15, pp. 113–120, 2014.
- [87] S. Khurshid, A. K. Shrivastava, and J. Iqbal, "Effort based software reliability model with fault reduction factor, change point and imperfect debugging," *International Journal of Information Technology*, Feb 2019.

- [88] S. T. Lai, “An analyzer-based software security measurement model for enhancing software system security,” *Proceedings - 2010 2nd WRI World Congress on Software Engineering, WCSE 2010*, vol. 2, pp. 93–96, 2010.
- [89] S. Conte, *Software engineering metrics and models*. Menlo Park, Calif: Benjamin/Cummings Pub. Co, 1986.
- [90] B. Alshammari, C. Fidge, and D. Corney, “A Hierarchical Security Assessment Model for Object-Oriented Programs,” *2011 11th International Conference on Quality Software*, no. 1, pp. 218–227, 2011.
- [91] R. T. Colombo, M. S. Pessôa, A. C. Guerra, A. B. Filho, and C. C. Gomes, “Prioritization of software security intangible attributes,” *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, p. 1, 2012.
- [92] S. Zafar, M. Mehboob, A. Naveed, and B. Malik, “Security quality model: an extension of Dromey’s model,” *Software Quality Journal*, vol. 23, no. 1, 2015.
- [93] S. Sentilles, E. Papatheocharous, and F. Ciccozzi, “What do we know about software security evaluation? A preliminary study,” in *6th International Workshop on Quantitative Approaches to Software Quality*, 12 2018.
- [94] N. M. Mohammed, M. Niazi, M. Alshayeb, and S. Mahmood, “Exploring Software Security Approaches in Software Development Lifecycle: A Systematic Mapping Study,” *Computer Standards & Interfaces*, vol. 50, no. October 2016, pp. 107–115, 2016.
- [95] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, “Software vulnerability prediction using text analysis techniques,” *Proceedings of the 4th international workshop on Security measurements and metrics - MetriSec '12*, p. 7, 2012.
- [96] M. Gegick and L. Williams, “Toward the use of automated static analysis alerts for early identification of vulnerability- and attack-prone components,” *Second International Conference on Internet Monitoring and Protection, ICIMP 2007*, no. Icimp, 2007.
- [97] T. Gyimothy, R. Ferenc, and I. Siket, “Empirical validation of object-oriented metrics on open source software for fault prediction,” *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

- [98] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, “Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly Iterative or agile software development processes,” *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 402–419, 2007.
- [99] D. Radjenović, M. Heričko, R. Torkar, and A. Živković, “Software fault prediction metrics: A systematic literature review,” *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.
- [100] L. Kumar, S. Misra, and S. K. Rath, “An empirical analysis of the effectiveness of software metrics and fault prediction model for identifying faulty classes,” *Computer Standards & Interfaces*, vol. 53, no. September 2016, pp. 1–32, 2017.
- [101] M. Child, P. Rosner, and S. Counsell, “A comparison and evaluation of variants in the coupling between objects metric,” *Journal of Systems and Software*, vol. 151, pp. 120 – 132, 2019.
- [102] P. J. Morrison, R. Pandita, X. Xiao, R. Chillarege, and L. Williams, “Are vulnerabilities discovered and resolved like other defects?,” *Empirical Software Engineering*, vol. 23, pp. 1383–1421, Jun 2018.
- [103] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, “Software metrics as indicators of security vulnerabilities,” in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 216–227, Oct 2017.
- [104] C. J. Clemente, F. Jaafar, and Y. Malik, “Is predicting software security bugs using deep learning better than the traditional machine learning algorithms?,” in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 95–102, July 2018.
- [105] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, “Predicting vulnerable software components,” *Proceedings of the 14th ACM conference on Computer and communications security CCS 07*, p. 529, 2007.
- [106] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, “Predicting vulnerable software components via text mining,” *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

- [107] Y. Pang, X. Xue, and A. S. Namin, "Predicting Vulnerable Software Components through N-Gram Analysis and Statistical Feature Selection," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pp. 543–548, Dec 2015.
- [108] Y. Pang, X. Xue, and H. Wang, "Predicting Vulnerable Software Components through Deep Neural Network," *Proceedings of the 2017 International Conference on Deep Learning Technologies - ICDLT '17*, pp. 6–10, 2017.
- [109] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li, "Combining Software Metrics and Text Features for Vulnerable File Prediction," *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS*, vol. 2016-Janua, pp. 40–49, 2016.
- [110] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pp. 23–33, 2014.
- [111] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing Software Security Fortification through Code-Level Metrics," *Proceedings of the 4th ACM workshop on Quality of Protection*, pp. 31–38, 2008.
- [112] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with Applying Vulnerability Prediction Models," *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, pp. 4:1–4:9, 2015.
- [113] J. Yang, D. Ryu, and J. Baik, "Improving vulnerability prediction accuracy with Secure Coding Standard violation measures," *2016 International Conference on Big Data and Smart Computing, BigComp 2016*, pp. 115–122, 2016.
- [114] "LDRA Tool Suite." [Online]. Available: <http://www.ldra.com/en/software-quality-test-tools/group/by-product-module/ldra-tool-suite>. Last Accessed: 01-03-2019.
- [115] M. C. Olesen, R. R. Hansen, J. L. Lawall, and N. Palix, "Clang and Coccinelle: Synergising program analysis tools for CERT C Secure Coding Standard certification," *Electronic Communications of the EASST*, vol. 33, no. 0, 2010.

- [116] M. C. Olesen, R. R. Hansen, J. L. Lawall, and N. Palix, “Coccinelle: Tool support for automated CERT C Secure Coding Standard certification,” in *Science of Computer Programming*, vol. 91, pp. 141–160, 2014.
- [117] Y. Shin and L. Williams, “Can traditional fault prediction models be used for vulnerability prediction?,” *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013.
- [118] M. Jimenez, Y. Le Traon, and M. Papadakis, “[engineering paper] enabling the continuous analysis of security vulnerabilities with vuldata7,” in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 56–61, Sep. 2018.
- [119] T. Boland and P. E. Black, “Juliet 1.1 C/C++ and java test suite,” *Computer (Long. Beach. Calif.)*, vol. 45, no. 10, pp. 88–90, 2012.
- [120] P. E. Black, “A software assurance reference dataset: Thousands of programs with known bugs,” *Journal of research of the National Institute of Standards and Technology*, vol. 123, 2018.
- [121] “OWASP Benchmark.” [Online]. Available: <https://www.owasp.org/index.php/Benchmark>. Last Accessed: 01-03-2019.
- [122] N. Losada, M. J. Martín, G. Rodríguez, and P. Gonzalez, “Portable application-level checkpointing for hybrid MPI-OpenMP applications,” *Procedia Computer Science*, vol. 80, pp. 19–29, 2016.
- [123] E. Gelenbe, “A Model of Roll-back Recovery with Multiple Checkpoints,” in *Proceedings of the 2Nd International Conference on Software Engineering, ICSE '76*, (Los Alamitos, CA, USA), pp. 251–255, IEEE Computer Society Press, 1976.
- [124] E. Gelenbe and D. Derochette, “Performance of Rollback Recovery Systems Under Intermittent Failures,” *Commun. ACM*, vol. 21, no. 6, pp. 493–499, 1978.
- [125] E. Gelenbe and M. Hernandez, “Enhanced availability of transaction oriented systems using failure tests,” *Software Reliability Engineering, 1992. Proceedings., Third International Symposium on*, pp. 342–350, 1992.

- [126] E. Gelenbe and M. Hernández, “Virus tests to maximize availability of software systems,” *Theoretical Computer Science*, vol. 125, no. 1, pp. 131–147, 1994.
- [127] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, “A Survey of Rollback-recovery Protocols in Message-passing Systems,” *ACM Comput. Surv.*, vol. 34, pp. 375–408, sep 2002.
- [128] G. Rodríguez, M. J. Martín, P. González, J. Touriño, and R. Doallo, “CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications,” *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 749–766, 2010.
- [129] E. Dijkstra, J. Buxton, and B. Randell, “Software Engineering Techniques,” *NATO Science Committee*, 1969.
- [130] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezze, “Automatic recovery from runtime failures,” *Proceedings - International Conference on Software Engineering*, pp. 782–791, 2013.
- [131] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, *Automatic Workarounds: Exploiting the Intrinsic Redundancy of Web Applications*, vol. 24. 2015.
- [132] A. Armoush, F. Salewski, and S. Kowalewski, “A hybrid fault tolerance method for recovery block with a weak acceptance test,” *Proceedings of The 5th International Conference on Embedded and Ubiquitous Computing, EUC 2008*, vol. 1, pp. 484–491, 2008.
- [133] E. Gelenbe and I. Mittrani, “Modelling the behaviour of block structured processes with hardware and software failures,” in *Mathematical Computer Performance and Reliability* (e. a. Giuseppe Iazeolla, ed.), pp. 329–339, Elsevier Science Publishers, 1984.
- [134] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, “Automatic Workarounds for Web Applications,” in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10*, (New York, NY, USA), pp. 237–246, ACM, 2010.
- [135] M. Uva, P. Ponzio, G. Regis, N. Aguirre, and M. F. Frias, “Automated Workarounds from Java Program Specifications Based on SAT Solving,” in *Fundamental Approaches to Software Engineering* (M. Huisman and J. Rubin, eds.), (Berlin, Heidelberg), pp. 356–373, Springer Berlin Heidelberg, 2017.

- [136] J. Duell, P. Hangrove, and E. Roman, “The design and implementation of berkeley lab’s linux checkpoint/restart,” *Berkeley Lab Technical Report (publication LBNL-54941)*, 2002.
- [137] J. Ansel, K. Arya, and G. Cooperman, “DMTCP: Transparent checkpointing for cluster computations and the desktop,” *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, 2009.
- [138] J. S. Plank, M. Beck, G. Kingsley, and K. Li, “Libckpt: Transparent Checkpointing Under Unix,” in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON’95, (Berkeley, CA, USA), p. 18, USENIX Association, 1995.
- [139] M. Litzkow, T. Tannenbaum, and M. Linvy, “Checkpoint and migration of UNIX processes in the Condor distributed processing system,” *Technical Report CS-TR- 199701346*, *University of Wisconsin, Madison*, 1997.
- [140] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi, “CheCL: Transparent checkpointing and process migration of OpenCL applications,” *Proceedings - 25th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2011*, pp. 864–876, 2011.
- [141] H. Takizawa, K. Sato, K. Komatsut, and H. Kobayashit, “CheCUDA: A checkpoint/restart tool for CUDA applications,” *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings*, pp. 408–413, 2009.
- [142] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, “The Design and Implementation of Checkpoint / Restart Process Fault Tolerance for Open MPI ,” *Architecture*, 2007.
- [143] M. Hasan and M. S. Goraya, “Fault tolerance in cloud computing environment: A systematic survey,” *Computers in Industry*, 2018.
- [144] “National Institute of Standards and Technology (NIST).” [Online]. Available: <https://www.nist.gov/>. Last Accessed: 01-03-2019.
- [145] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “A view of cloud computing,” *Commun. ACM*, vol. 53, pp. 50–58, Apr. 2010.

- [146] B. Nicolae and F. Cappello, “Blobcr: Efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, (New York, NY, USA), pp. 34:1–34:12, ACM, 2011.
- [147] D. Sun, G. Chang, C. Miao, and X. Wang, “Modelling and evaluating a high serviceability fault tolerance strategy in cloud computing environments,” *International Journal of Security and Networks*, vol. 7, no. 4, pp. 196–210, 2012.
- [148] J. Cao, M. Simonin, G. Cooperman, and C. Morin, “Checkpointing as a service in heterogeneous cloud environments,” in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 61–70, May 2015.
- [149] F. Shahzad, M. Wittmann, T. Zeiser, and G. Wellein, “Asynchronous Checkpointing by Dedicated Checkpoint Threads,” in *Recent Advances in the Message Passing Interface* (J. L. Träff, S. Benkner, and J. J. Dongarra, eds.), (Berlin, Heidelberg), pp. 289–290, Springer Berlin Heidelberg, 2012.
- [150] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, “Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System,” in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, nov 2010.
- [151] M. Siavvas, D. Kehagias, and D. Tzovaras, “A preliminary study on the relationship among software metrics and specific vulnerability types,” *International Conference on Computational Science and Computational Intelligence (CSCI) 2017*, pp. 916–921, 2017.
- [152] “Maven Repository.” [Online]. Available: <https://mvnrepository.com/>.
- [153] S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object Oriented Design,” *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, 1994.
- [154] “CKJM Extended.” [Online]. Available: http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/. Last Accessed: 01-03-2019.
- [155] “PMD Static Code Analyzer.” [Online]. Available: <https://pmd.github.io/>. Last Accessed: 01-03-2019.

- [156] G. B. de Pádua and W. Shang, “Studying the prevalence of exception handling anti-patterns,” in *Proceedings of the 25th International Conference on Program Comprehension*, ICPC ’17, (Piscataway, NJ, USA), pp. 328–331, IEEE Press, 2017.
- [157] G. B. de Pádua and W. Shang, “Studying the relationship between exception handling practices and post-release defects,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR ’18, (New York, NY, USA), pp. 564–575, ACM, 2018.
- [158] C. Spearman, “The proof and measurement of association between two things. By C. Spearman, 1904.,” *The American journal of psychology*, vol. 100, no. 3-4, pp. 441–471, 1987.
- [159] J. Cohen, *Statistical power analysis for the behavioral sciences*. Hillsdale, N.J: L. Erlbaum Associates, 1988.
- [160] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.
- [161] M. Howard, *Writing secure code*. Redmond, Wash: Microsoft Press, 2003.
- [162] T. L. Saaty, “Decision making with the analytic hierarchy process,” *International Journal of Services Sciences*, vol. 1, no. 1, p. 83, 2008.
- [163] J. Andress, *The basics of information security : understanding the fundamentals of InfoSec in theory and practice*. Waltham, MA: Syngress, 2014.
- [164] *SP 800-160: Systems Security Engineering Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems*. National Institute of Standards and Technology.
- [165] ISO/IEC, “ISO/IEC 27001:2013(en) Information technology Security techniques Information security management systems Requirements,” tech. rep., 2013.
- [166] W. Edwards and F. Barron, “SMARTS and SMARTER: Improved Simple Methods for Multiattribute Utility Measurement,” *Organizational Behavior and Human Decision Processes*, vol. 60, no. 3, pp. 306–325, 1994.

- [167] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” *ACM SIGPLAN Notices*, vol. 39, no. 12, p. 92, 2004.
- [168] K. Goseva-Popstojanova and A. Perhinschi, “On the capability of static code analysis to detect security vulnerabilities,” *Information and Software Technology*, vol. 68, pp. 18–33, 2015.
- [169] M. Siavvas and E. Gelenbe, “Optimum intervals for application-level checkpoints,” in *2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, 2019.
- [170] M. Siavvas and E. Gelenbe, “Optimum checkpoints for programs with loops,” *Simulation Modelling Practice and Theory*, 2019.
- [171] “Summary of the Amazon s3 service disruption in the northern Virginia (us-east-1) region,” February 2018.
- [172] “Top 20 high profile cloud failures of all time,” *TechFlier*.
- [173] C. Wueest, M. B. Barcena, and L. O’Brien, “Mistakes in the IaaS cloud could put your data at risk,” *Symantec*.
- [174] E. Gelenbe, “Dealing with software viruses: a biological paradigm,” *Information Security Technical Report*, vol. 12, no. 4, pp. 242–250, 2007.
- [175] E. Gelenbe, “Model on information renewal by the method of multiple test-points (english translation of *avtomat. i telemekh.*, 1979:4, pp. 142-151,” *Automation and Remote Control*, vol. 40, no. 4, pp. 598–605, 1979.
- [176] E. Gelenbe and G. Pujolle, *Introduction aux Réseaux de Files d’Attente*. Editions Hommes et Techniques, Eyrolles, 1982.
- [177] E. Gelenbe and I. Mitrani, *Analysis and synthesis of computer systems*. World Scientific, 2010.
- [178] E. Gelenbe, D. Finkel, and S. K. Tripathi, “Availability of a distributed computer system with failures,” *Acta Informatica*, vol. 23, p. 643, 1986.

- [179] S. K. Tripathi, D. Finkel, and E. Gelenbe, "Load sharing in distributed systems with failures," *Acta Inf.*, vol. 25, no. 6, pp. 677–689, 1988.
- [180] D. Dauwe, R. Jhaveri, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, "Optimizing checkpoint intervals for reduced energy use in exascale systems," in *Eighth International Green and Sustainable Computing Conference, IGSC 2017, Orlando, FL, USA, October 23-25, 2017*, pp. 1–8, 2017.
- [181] F. Kong, M. Xu, J. Weimer, O. Sokolsky, and I. Lee, "Cyber-physical system checkpointing and recovery," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS '18, (Piscataway, NJ, USA)*, pp. 22–31, IEEE Press, 2018.
- [182] M. Cinque, S. Russo, C. Esposito, K. R. Choo, F. Free-Nelson, and C. A. Kamhoua, "Cloud reliability: Possible sources of security and legal issues?," *IEEE Cloud Computing*, vol. 5, pp. 31–38, May 2018.
- [183] R. C. Seacord, *Secure Coding in C and C++*. Pearson Education, 2005.
- [184] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code," *Proceedings - Annual Computer Security Applications Conference, ACSAC*, pp. 257–267, 2000.
- [185] M. Rigger, "Sandboxed execution of c and other unsafe languages on the java virtual machine," in *Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming, Programming'18 Companion, (New York, NY, USA)*, pp. 227–229, ACM, 2018.
- [186] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.
- [187] N. Antunes and M. Vieira, "On the Metrics for Benchmarking Vulnerability Detection Tools," in *Proceedings of the International Conference on Dependable Systems and Networks*, vol. 2015-Sept, 2015.
- [188] P. Ffleeger and S. L. Pfleeger, *Security in Computing*, vol. 2nd. 2006.

- [189] J. Luszcz, “Apache struts 2: how technical and development gaps caused the equifax breach,” *Network Security*, vol. 2018, no. 1, pp. 5 – 8, 2018.
- [190] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler, “Heartbleed 101,” *IEEE Security Privacy*, vol. 12, pp. 63–67, July 2014.
- [191] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. a. S. E. I. T. on Vouk, “On the value of static analysis for fault detection in software,” *Software Engineering, IEEE Transactions on*, vol. 32, no. 4, pp. 240–253, 2006.
- [192] W. G. J. Halfond, J. Viegas, and A. Orso, “A Classification of SQL Injection Attacks and Countermeasures.” 2008.
- [193] S. Fogie, J. Grossman, R. Hansen, a. Rager, and P. Petkov, *XSS Attacks: Cross Site Scripting Exploits and Defense*. 2007.
- [194] OWASP, “OWASP Top 10 - 2013,” *OWASP Top 10*, p. 22, 2013.
- [195] CWE, “CWE - 2011 cwe/sans top 25 most dangerous software errors,” *SANS Institute*, p. 41, 2011.
- [196] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: a static analysis tool for detecting Web application vulnerabilities,” in *2006 IEEE Symposium on Security and Privacy (S&P’06)*, pp. 6 pp.–263, 2006.
- [197] M. K. Gupta, M. C. Govil, G. Singh, and P. Sharma, “XSSDM: Towards detection and mitigation of cross-site scripting vulnerabilities in web applications,” *2015 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2015*, vol. 1, pp. 2010–2015, 2015.
- [198] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps,” *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI ’14*, pp. 259–269, 2013.
- [199] I. Medeiros, N. Neves, and M. Correia, “Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining,” *IEEE Transactions on Reliability*, vol. 65, no. 1, 2016.

- [200] A. Austin and L. Williams, “One technique is not enough: A comparison of vulnerability discovery techniques,” in *International Symposium on Empirical Software Engineering and Measurement*, pp. 97–106, 2011.
- [201] A. Austin, C. Holmgreen, and L. Williams, “A comparison of the efficiency and effectiveness of vulnerability discovery techniques,” *Information and Software Technology*, vol. 55, no. 7, pp. 1279–1288, 2013.
- [202] R. Scandariato, J. Walden, and W. Joosen, “Static analysis versus penetration testing: A controlled experiment,” *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 451–460, 2013.
- [203] B. Chess, *Secure programming with static analysis*. Upper Saddle River, NJ: Addison-Wesley, 2007.
- [204] N. Ayewah and W. Pugh, “The Google FindBugs fixit,” in *Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10*, p. 241, 2010.
- [205] V. B. Livshits and M. S. Lam, “Finding Security Vulnerabilities in Java Applications with Static Analysis,” *Architecture*, p. 18, 2005.

Appendix A

Static Analysis

Static analysis constitutes an important concept of the present thesis, since the studied methods are either directly or indirectly related to static analysis. As far as software security is concerned, automated static analysis (ASA) tools are normally used to detect security issues in software products, and therefore their outputs constitute valuable sources of security-relevant information, which could potentially be exploited either for security assessment or for vulnerability prediction. With respect to software reliability, Application-level Checkpoint and Restart (ALCR) libraries and tools normally use static analysis in order to detect the locations in which checkpoint should be inserted, as well as for automating the insertion of the checkpoints in the applications by automatically modifying their source code. The purpose of the present section is to familiarize the reader with the concept of static analysis, by providing some important background concepts. The importance of ASA tools in secure software development is also discussed, since their outputs constitute the basis for the construction of the Security Assessment Model (SAM) presented in Chapter 4.

A.1 Background Concepts

Static analysis is a testing technique that searches for potential software bugs (including vulnerabilities) by analyzing the product source or compiled code, without requiring its execution [1]. Hence, it can be applied at any point of the software development cycle, regardless of whether all the components of the overall product are properly operating or even successfully compiling.

In general, two types of static analysis exist [1, 9]: (i) *manual code inspection*, which requires the inspection of the code by one or more software engineers, and (ii) *automated static analysis (ASA)*, which is executed autonomously, without requiring human interaction. In fact, the purpose of the latter is to automate the execution of static analysis, and also to increase the effectiveness of manual code reviewing, which is considered to be a highly time-consuming and error-prone testing technique [7, 9, 15]. The terms static analysis and automated static analysis are often used interchangeably in the related literature.

Automated static analysis is performed via automated static analysis tools. The output of ASA tools is usually a list of warnings (i.e. alerts), which correspond to potential issues (e.g. vulnerabilities)¹ [20]. This list should be manually inspected to determine which of the produced alerts correspond to actual bugs and, thus, require immediate attention. The inspection is necessary, as the main problem of the tools is the production of false positives and false negatives. False positives refer to alerts that correspond to non-existing software bugs, while false negatives refer to existing issues that the tool fails to report [7].

False positives and false negatives are generated due to the approximations that the tools are obliged to make in order to achieve satisfactory scalability and performance [9, 20]. In fact, the analysis is not applied directly to the actual code, but on a model generated from the analyzed source code. Two different techniques are used for the derivation of the model, namely (i) under-approximation, and (ii) over-approximation. In over-approximation, more states than the minimum required are used in order to represent the source code, leading to the identification of more actual bugs, but also in the generation of more false positives. In under-approximation, fewer states are used to describe the source code, leading to fewer false positives, but in an increased possibility of missing actual issues (i.e., introduction of false negatives). Over-approximation is widely used in the production of ASA tools, as omitting actual bugs is considered a more severe problem than reporting false issues [187].

Another factor that influences the production of false positives and false negatives is the amount of code context (i.e., code semantics) that the tool takes into consideration during the analysis [20]. More context is better when it comes to reducing false positives, but it can lead to a huge amount of computation to perform [9, 15]. This indicates that the development of an ideal tool,

¹Other outputs of static analysis tools are software metrics. However, the description of software metrics tools is out of the scope of the present section.

i.e., a tool able to detect all the existing issues, by maintaining a satisfactory performance and without generating false positives, is feasible only in theory [7]. Hence, in practice, the most effective tool is the one that strikes the right balance among false positives, false negatives, and performance [7].

A.2 Static Analysis for Security

Vulnerabilities are special types of software bugs with security implications [73]. They lead to the violation of one or more security policies, normally without influencing the functional requirements of the software products, constituting in that way their detection by conventional functional testing mechanisms highly difficult (if not impossible). According to [1], two types of software vulnerabilities exist: (i) *design-level vulnerabilities*, which may occur as a result of a design flaw, and (ii) *implementation-level vulnerabilities*, which may occur as a result of a bug in the code. A *design-level vulnerability* often manifest itself as a flaw in the code of the system (i.e., *implementation-level vulnerability*), if it is not eliminated during the early stages of the SDLC [188]. Therefore, as already mentioned, most of the software vulnerabilities stem from a small number of common programming errors [9,10]. On average, roughly half of the software vulnerabilities are introduced during the coding phase [1]. It should be noted that well-known security breaches like the Equifax Breach [189] and the Heartbleed [190] were caused by simple implementation errors.

The majority of the issues that can be found by static analysis are common programming errors [191]. Since software vulnerabilities are normally caused by common programming errors, static analysis is considered a highly suitable and effective technique for detecting security vulnerabilities that reside in the source code of software products. In fact, ASA tools have been found effective in identifying security-related bugs early enough in the SDLC [9,15], when their correction is relatively cheap and easy [11]. For instance, the most representative example of vulnerabilities that can be detected by ASA tools is injection vulnerabilities (e.g. SQL Injection (SQLI) [192] and Cross-site Scripting (XSS) [193]), which are considered as the most common and dangerous threats of web applications according to both OWASP Top 10 [194] and CWE Top 25 [195] lists of most common software vulnerabilities. These vulnerabilities are commonly caused by unsanitized data flows from untrusted inputs to security sensitive sinks [20], which

is a relatively simple but all too often overlooked code-level error. A multitude of ASA tools have been proposed for detecting injection vulnerabilities (e.g. [167, 196–199]), indicating the importance of static analysis in detecting such types of security issues.

Apart from the fact that ASA tools are able to detect the most common software vulnerabilities, there are also other factors that highlight their importance in secure software development. According to several experts in the field of software security, automated static analysis is the most effective technique for adding security during software development [15] and, thus, it should be applied regularly as part of any modern development process [9]. In addition, most of the well-established secure SDLCs that have been proposed over the years, such as Microsoft’s SDL [17, 161], OWASP’s OpenSAAM², and Cigital’s Touchpoints [1], propose the adoption of static analysis as the major mechanism for adding security during the coding (i.e., implementation) phase of the SDLC. Moreover, ASA is one of the most common security activities adopted in practice, as reported by BISIMM³, which is a knowledge base that contains common security activities adopted by 109 major technological firms, including Google, Microsoft, Adobe, and Intel. Finally, automated static analysis has been widely used in the literature for security purposes, including vulnerability detection (e.g., [167, 196–199]), prediction (e.g., [66, 96, 111, 113]), and automatic elimination (e.g., [199]) with promising results. According to a recent systematic mapping study in the field of Security by Design [94], vulnerability detection, prediction and elimination using static analysis is the current trend in the field.

Static analysis is believed to be more effective in detecting code-level vulnerabilities compared to other dynamic vulnerability detection mechanisms (VDMs) like penetration testing [18] and fuzzing [19], as it is observed to produce significantly fewer false negatives [20]. This can be explained by the fact that software vulnerabilities often exist in states that are hard-to-reach, and that ASA tools can peer into these states more efficiently than dynamic analysis [9]. However, several empirical studies revealed that one single technique is not enough for identifying all the vulnerabilities that a software product contains [200–202]. In fact, different VDMs are able to detect different and usually independent types of software vulnerabilities. For instance, static analysis is more effective in detecting implementation bugs, while penetration testing is more capable in identifying design flaws [201]. Therefore, in order to discover the greatest variety of vulnerability types, different VDMs should be employed complementarily.

²<http://www.opensamm.org/>

³<https://www.bsimm.com/>

Several ASA tools have been proposed over the years for finding code-level security issues in software products, with the main purpose to achieve a satisfactory trade-off among false positives, false negatives, and performance. Another important requirement of the developed tools is that they should encapsulate most of the required security expertise, so that they can be easily used even by developers that are not security experts [9, 12, 203]. For reasons of brevity, only the most representative examples of these tools are presented in the following. A detailed survey of static code analyzers for security auditing can be found in [20, 203].

Early ASA tools were based on lexical analysis for detecting common security issues. In this approach, the source code of the software product under analysis is preprocessed and tokenized, and then the resulting token stream is matched against a library of vulnerable constructs. The most representative examples of tools that utilize lexical analysis are ITS4 [184], FlawFinder⁴, and RATS⁵. ITS4 is believed to be the first static code analyzer for security auditing. It was created by Cigital in order to help developers of C and C++ programs avoid common security-related coding issues. It detects common security issues, by scanning through source code files looking for syntactic matches based on several simple “rules” that might indicate possible security vulnerabilities (for example, “the use of strcpy() should be avoided”). ITS4 was used as the basis for the creation of Fortify SCA [69], a highly accurate and precise commercial source code analyzer for detecting code-level security issues, which is widely used in the related literature (e.g., [106, 107]). FlawFinder and RATS are highly similar to ITS4. Their main difference is that they are able to detect different almost non-overlapping classes of vulnerabilities, while RATS can also analyze source code written in programming languages other than C/C++, including Perl, PHP, and Python. However, the main problem of these early lexical analysis-based tools is that they produce a large number of false positives, as they totally neglect the semantics (i.e., context) of the analyzed code.

To reduce the number of the produced false positives, semantic analysis was adopted by next generation tools. In this approach, the semantics of the source code (e.g., data flows and control flows) are taken into consideration during static analysis. PMD [155], Findbugs [167], and WAP [199] are three characteristic tools of this category. PMD [155] builds the Abstract Syntax Tree (AST) of the analyzed code in order to catch the basic semantics of the source code. Subsequently, it applies pattern matching on the extracted model in order to identify

⁴<https://www.dwheeler.com/flawfinder/>

⁵<https://code.google.com/archive/p/rough-auditing-tool-for-security/>

violations of specific patterns (called rules) that correspond to best coding practices. Newer versions of the tool also support data-flow analysis in order to improve the overall precision. Findbugs [167] is an open-source static code analyzer that analyzes the compiled bytecodes of Java programs by employing both data-flow and control-flow analysis to detect bugs, including vulnerabilities. It is widely used in the related literature for security auditing purposes, while Google has also incorporated it into its software development process [204]. WAP [199] is an ASA tool that semantically analyzes the source code of web applications written in PHP in order to detect input validation vulnerabilities, like XSS and SQLI. In particular, it performs taint analysis (data-flow analysis) to examine whether untrusted inputs inserted by the systems entry points reach security-sensitive sinks. Data mining is employed to discriminate actual bugs from false positives, while actual bugs are automatically fixed by the tool.

Context sensitivity was also employed in an attempt to further reduce the number of the produced false positives. Several tools have been proposed in the literature adopting this concept. Lisvits and Lam [205] demonstrated how context-sensitive pointer analysis together with string analysis can facilitate the detection of injection vulnerabilities, with a highly small number of false positives. In [197], Gupta et al. proposed XSSDM, a context-sensitive approach based on static taint analysis and pattern matching for detecting XSS vulnerabilities in the source code of web applications written in PHP. The evaluation results indicate that the proposed approach outperforms Pixy [196] both in terms of accuracy and precision, which is another popular context-sensitive taint analyzer for detecting injection vulnerabilities in PHP, with relatively low false positive rate. Finally, FlowDroid [198] is the most complete static taint analysis tool so far that adopts context-, flow-, field-, and object-sensitivity in order to detect injection vulnerabilities in Android applications. FlowDroid significantly outperforms well-known commercial tools like AppScan⁶ and Fortify SCA [69] in terms of accuracy and precision.

Despite their promising results in false positive reduction, context-sensitivity-based tools are hindered by poor performance and scalability. The more context a tool incorporates, the more expensive its analysis is in terms of memory and execution time [21]. In addition, these tools are usually effective in detecting only a small subset of security issues (e.g., injection vulnerabilities). This mandates the adoption of multiple tools in order to cover a broad spectrum of vulnerability types, which may be unacceptable in terms of memory and time required for the conduction

⁶<https://www.ibm.com/security/application-security/appscan>

of the overall security auditing. However, all these tools provide a large volume of outputs that contain security-relevant information, which could potentially be exploited for extracting security knowledge that could be useful for secure software development. For instance, they could be used as the basis for software security assessment as well as for vulnerability prediction.