Software Quality Journal
https://doi.org/10.1007/s11219-021-09555-0

A hierarchical model for quantifying software security based on static analysis alerts and software metrics

Miltiadis Siavvas[1,2] · Dionysios Kehagias[2] · Dimitrios Tzovaras[2] · Erol Gelenbe[1,3]

Accepted: 23 March 2021

Abstract
Despite the acknowledged importance of quantitative security assessment in secure software development, current literature still lacks an efficient model for measuring internal software security risk. To this end, in this paper, we introduce a hierarchical security assessment model (SAM), able to assess the internal security level of software products based on low-level indicators, i.e., security-relevant static analysis alerts and software metrics. The model, following the guidelines of ISO/IEC 25010, and based on a set of thresholds and weights, systematically aggregates these low-level indicators in order to produce a high-level security score that reflects the internal security level of the analyzed software. The proposed model is practical, since it is fully automated and operationalized in the form of a standalone tool and as part of a broader Computer-Aided Software Engineering (CASE) platform. In order to enhance its reliability, the thresholds of the model were calibrated based on a repository of 100 popular software applications retrieved from Maven Repository. Furthermore, its weights were elicited in a way to chiefly reflect the knowledge expressed by the Common Weakness Enumeration (CWE), through a novel weights elicitation approach grounded on popular decision-making techniques. The proposed model was evaluated on a large repository of 150 open-source software applications retrieved from GitHub and 1200 classes retrieved from the OWASP Benchmark. The results of the experiments revealed the capacity of the proposed model to reliably assess internal security at both product level and class level of granularity, with sufficient discretion power. They also provide preliminary evidence for the ability of the model to be used as the basis for vulnerability prediction. To the best of our knowledge, this is the first fully automated, operationalized and sufficiently evaluated security assessment model in the modern literature.

Keywords Software Security · Software Quality Evaluation · Security Assessment

✉ Miltiadis Siavvas
m.siavvas16@imperial.ac.uk

Extended author information available on the last page of the article

Published online: 15 May 2021

1 Introduction

Measuring software security is very important for secure software development, since it actually quantifies the success of the applied security policies (Hahn et al., 2018; Rindell, et al. 2019), such as Microsoft's SDL (Howard, 2003; Howard and Lipner, 2006). In addition, it is a common belief that *"you cannot control something you cannot measure"* (DeMarco, 1986). Although mature and mutually accepted measurements of external security exist (e.g., the *Attack Surface Metric* (Manadhata and Wing, 2011; Hahn et al., 2018)), current literature still lacks a commonly accepted approach for evaluating the internal security risk of software products[1] (Ansar et al., 2018; Sentilles et al., 2018; Morrison et al., 2018). Different metrics and assessment methodologies able to measure a specific security aspect (Chowdhury et al., 2008; Alshammari et al., 2009; Shin et al., 2014), or the complete security level of software products (Lai, 2010; Alshammari et al., 2011; Medeiros et al., 2018; Dayanandan, 2018) 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 (Verendel, 2009; Sentilles et al., 2018; Ansar et al., 2018; Morrison et al., 2018).

Software security assessment is commonly treated as a subfield of software quality evaluation (Munaiah et al., 2017). However, although a large number of quality models have been proposed for evaluating specific quality attributes, like *Maintainability* (Heitlager, et al. 2007; Bakota et al., 2011; Baggen et al., 2012; di Base et al., 2019) and *Reliability* (Jin & Jin, 2014; Khurshid et al., 2019), 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 (ISO, 2011) international standard on software quality. 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 extended) for building reliable security assessment models (Basso et al., 2019).

The main objective of a security assessment model is to provide a quantifiable expression of software security by combining several heterogeneous security metrics (Verendel, 2009). Static analysis alerts and software metrics constitute valuable sources of security information, and therefore promising candidates for the construction of security assessment models. In fact, static analysis, a testing approach that allows the identification of software bugs without requiring code execution (Chess & McGraw, 2004), has been found effective in uncovering vulnerabilities (McGraw, 2006; Nunes et al., 2019), whereas several studies have highlighted the ability of software metrics (e.g., complexity) to indicate the existence of vulnerabilities in software products (e.g., (Shin et al., 2011; Medeiros et al., 2017)). 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 (e.g., (Zafar et al., 2015; Medeiros et al., 2018; Dayanandan and Kalimuthu, 2018)).

To this end, in this paper, we introduce 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

[1] It should be noted that the term software security is used to describe the internal security of software products, while the external security is often termed as application security (McGraw, 2006).

ISO/IEC 25010 (ISO, 2011), 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.

The proposed model is highly practical, since it is operationalized, both as a standalone tool (Online, 2020) and as part of the SDK4ED Platform (see Section 4), and fully automated, and therefore it can be used regularly during the overall development cycle. In addition, its design parameters (i.e., *thresholds* and *weights*) were carefully elicited in order to avoid subjective information as much as possible[2]. More specifically, the thresholds were calibrated based on a benchmark repository of 100 Java applications retrieved from Maven Repository[3]. The weights of the model were elicited in a way to chiefly reflect the knowledge expressed by the Common Weakness Enumeration (CWE)[4], 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[5]. 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, whereas it is the only model that was built and evaluated on such a large volume of empirical data (i.e., 250 real-world software applications, comprising approximately 20 million lines of code). To facilitate the readability of the present paper, a roadmap of our work regarding the model construction and evaluation is depicted in Fig. 1.

In brief, the major contributions of the present paper 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; and (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.

At this point, it should be noted that the main research goal of the present paper is to investigate whether well-established concepts from the field of software quality evaluation can be leveraged for building models that can reliably assess software security. To this end, a carefully curated SAM for programs written in Java programming languages was derived and used as a proof-of-concept through extensive experimentation. However, since no single model is able to satisfy all the user and project needs (Siavvas et al., 2017b; Wagner et al., 2015), in the rest of the paper, we put specific emphasis on demonstrating in detail the internal structure and characteristics of the proposed model, as well as the steps that were followed for its construction, so that interested researchers or practitioners can build similar models that better meet their needs (i.e., types of software, programming languages, supported issues, etc.). To further facilitate this process, a similar model that

[2] With the term *subjective information* we refer to information that is defined based on the subjective opinions (i.e., judgments) of a limited number of individuals (i.e., usually the authors of the models) based on their security expertise, and not based on widely accepted sources of security information (e.g., international standards, security knowledge bases, etc.).

[3] https://mvnrepository.com/

[4] https://cwe.mitre.org/

[5] https://owasp.org/www-project-benchmark/

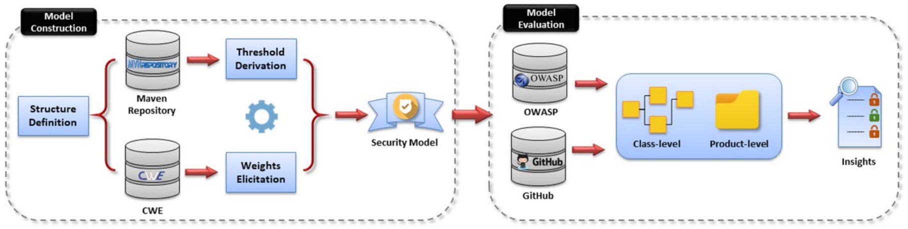


Fig. 1 The roadmap of the present paper. The work consists of two important steps, namely *Model Construction* and *Model Evaluation*. *Model Construction* is responsible for the definition of the model structure, as well as for the calculation of the main parameters of the proposed model (i.e., weights and thresholds). The *Model Evaluation* is responsible for evaluating the ability of the model to provide reliable security assessments at both product and class levels

was built as part of the SDK4ED Project for analyzing software products written in C/C++ programming language is also described in the Appendix section of the paper.

The rest of the paper is organized as follows: Section 2 presents the related work in the field of software security assessment. Section 3 introduces our model and describes its derivation steps. Section 4 presents the implementation of the model and its integration in a broader CASE platform. Section 5 reports the results of our experimental evaluation. Section 7 concludes the paper and discusses ideas for future work. Finally, a similar security model for evaluating the security level of software projects written in C and C++ programming languages is described in the Appendix section of the present paper.

2 Related work

This section provides related work in the field of software security assessment. Emphasis is given on the main open issues of the field, as well as on the major contributions of the present work.

2.1 Existing software security assessment approaches

The attack surface is the most complete and commonly accepted measure for quantifying the security level of software products that has been proposed in the literature until today. It was originally introduced by (Howard et al., 2005), who also proposed a method for its quantification called *Relative Attack Surface Quotient (RASQ)* (Howard, 2007). It was made fully operationalized by (Manadhata & Wing, 2011), through the proposal of the *Attack Surface Metric (ASM)*, which constitutes the *de facto* metric for its calculation. According to their approach, the attack surface of a software product is defined based on the resources that are used in attacks (i.e., entry/exit points, channels, and data), along with their possibility of being exploited, which is expressed by their *damage potential-effort (DP-E)*. This information is summarized into a single value, which corresponds to the attack surface of the application.

The main shortcoming of the attack surface metric that remains unresolved until today is that it neglects the innate characteristics of the software products that are known to influence their 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. Although several improvements of the *ASM* have been proposed over the years with the purpose to also consider the underlying security countermeasures (e.g., (Hatzivasilis et al., 2016; Hahn et al., 2018)) and the internal structure of the analyzed software (e.g., (Munaiah & Meenley, 2016)), the attack surface is still independent of the number of vulnerabilities that a software product contains (Theisen et al., 2018). Hence, to enhance the completeness of security assessment, attack surface metrics should be used in conjunction with measurements that take into account the internal characteristics of software products (Manadhata & Wing, 2011), such as (i) security-relevant software metrics, and (ii) vulnerability-based security measurements.

The ability of common software metrics to indicate security risks in software products has been widely studied in the literature (Siavvas et al., 2018a). (Shin & Williams, 2008) were the first to observe the ability of complexity metrics to indicate the existence of vulnerabilities in software components. In particular, they found that a statistically significant albeit weak correlation exists between software metrics and the existence of vulnerabilities in the Mozilla codebase. Similarly, (Chowdhury & Zulkernine, 2010) revealed the capacity of common coupling, cohesion, and complexity (CCC) metrics to discriminate between vulnerable and clean software components. These observations are supported by the results of a large number of follow-up empirical studies, which also proposed numerous software metrics-based vulnerability prediction models (Chowdhury & Zulkernine, 2011; Shin et al., 2011; Moshtari et al., 2013; Moshtari, & Sami 2016; Stuckman et al., 2017; Siavvas et al., 2017b; Ferenc et al., 2019; Jimenez et al., 2019). All these studies, which were based on different datasets, observed that a weak but statistically significant correlation exists between the software metrics and the existence of vulnerabilities, whereas they also produced metric-based vulnerability prediction models of satisfactory accuracy. In addition to this, recent studies have shown that the combination of different software metrics lead to better vulnerability predictors, and, thus, it may render a meaningful approach for enhancing security assessment (Medeiros et al., 2017; Zhang et al., 2019).

Apart from common software metrics, a small number of custom security metrics can be found in the related literature. For instance, (Chowdhury et al., 2008) proposed 3 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, (Alshammari et al., 2009) proposed a set of 7 design-level metrics for quantifying the security level (i.e., confidentiality) of object-oriented classes, from the viewpoint of potential information loss. The same authors proposed a new set of measures for assessing information-flow security of object-oriented designs (Alshammari et al., 2010). (Abdulrazeg et al., 2012) introduced a set of security metrics, promising to quantify and improve the misuse case model. (Shin et al., 2014) proposed an authenticity metric for measuring the degree to which an Android application is overprivileged, based on information retrieved from static code analysis. However, none of these metrics is operationalized in the form of a tool, while they are custom metrics that have not been extensively evaluated for their ability to reflect security concerns (Ansar et al., 2018).

As far as the vulnerability-based measurements are concerned, the vulnerability density metric is the most representative measure of this category. It was introduced by (Alhazmi et al., 2007) as a way to assess the overall security level of software products based on their reported vulnerabilities. It is defined as the ratio of the total number of the reported vulnerabilities that a product contains to the product size expressed in thousand lines of code. The *static analysis vulnerability density (SAVD)* (Walden et al., 2009) 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. It is used in the literature as a security indicator (e.g., (Walden & Doyle, 2012; Siavvas et al., 2017b; Siavvas et al., 2019b)), while it has been also used as the basis for the construction of vulnerability predictors (e.g., (Tang et al., 2015)).

The main problem of existing internal security metrics is that they are unable to reliably assess the overall security level of software products, since they quantify only a specific facet of software security (e.g., (Chowdhury et al., 2008; Alshammari et al., 2009; Shin et al., 2014)), or they attempt to evaluate the overall security based on limited information (e.g., (Alhazmi et al., 2007)). Hence, there is a need for a sophisticated approach, i.e., a security model, able to combine low-level security metrics in order to provide a reliable indicator of software security (Verendel, 2009). As already mentioned, security models can be derived based on quality models, due to the fact that software security was recently recognized as one of the main software quality characteristics, with the issuing of ISO/IEC 25010 (ISO, 2011) international standard on software quality. It should be noted that in a recent empirical study, (Basso et al., 2019) highlighted the ability of traditional quality models to quantify trustworthiness attributes, and mainly security.

To this end, and in order to expand the notions originally proposed for software quality evaluation into the security realm, a small number of security models have already been proposed. For instance, (Lai, 2010) proposed a security model that calculates the overall security of software products based on a set of custom statically and dynamically collected metrics. In fact, the overall score is the weighted average of the normalized metric values. Similarly, (Alshammari et al., 2011) proposed a hierarchical model for assessing the security (in fact, the confidentiality) of object-oriented programs from class-level custom metrics that quantify the potential flow of classified data between classes (that were proposed by (Alshammari et al., 2009; Alshammari et al., 2010)). Despite their usefulness, the reliability of these models is hindered since their design parameters are arbitrarily selected (i.e., not selected based on a formal model like ISO/IEC 25010), and they are also based on custom metrics that have not been sufficiently studied for their ability to indicate security risks. Moreover, the weights of the aggregation schemes were selected subjectively (i.e., based on their authors' judgments) and not based on any recognized source of information.

In an attempt for more reliable security assessment, (Xu et al., 2013) and (Colombo et al., 2012) proposed hierarchical models for measuring the security level of software products based on ISO/IEC 25010. According to their approach, software security is decomposed into a set of security characteristics as defined by ISO/IEC 25010, which are further decomposed into a set of more tangible properties. In both models, these properties need to be evaluated manually through code inspection by a group of experts. This comprises a major shortcoming, since they are not automated, and therefore they cannot be applied regularly during the development cycle. They are also characterized by subjectivity, since the low-level properties are assessed based on expert judgments (i.e., by their authors based on their security expertise).

Recently, several models have been proposed for measuring software security based exclusively on object-oriented (OO) metrics. For example, (Medeiros et al., 2018) proposed a *Trustworthiness Assessment Model*, in which the security of a software system is measured by taking the weighted average of the normalized values of a selected set of OO metrics. The final set of metrics, along with their associated weights, were determined based on their ability to predict software vulnerabilities, as reported by a previous empirical study (Medeiros et al., 2017). In another recent work, (Dayanandan & Kalimuthu, 2018) proposed a hierarchical model for assessing software security at architectural level, based exclusively on OO metrics retrieved from the QMOOD metric suite (Bansiya &

Davis, 2002). Similarly to the works of (Xu et al., 2013) and (Colombo et al., 2012), ISO/IEC 25010 was used for the definition of the model structure, whereas its parameters (i.e., weights) were defined based exclusively on expert judgments.

However, the reliability of these models is hindered since they are based exclusively on OO metrics, which were found to be only weak indicators of vulnerabilities (Shin & Williams, 2008; Chowdhury & Zulkernine, 2011; Shin et al., 2011; Siavvas et al., 2017; Siavvas et al., 2017b; Moshtari et al., 2013; Moshtari & Sami, 2016; Stuckman et al., 2017; Ferenc et al., 2019; Jimenez et al., 2019; Zhang et al., 2019). In addition, their parameters (i.e., thresholds, weights, etc.) were selected either based on very limited (Medeiros et al., 2018) or subjective information defined by their authors (Dayanandan & Kalimuthu, 2018) (i.e., expert judgments). They also lack sufficient evaluation, as both of them were evaluated through simple case studies that were based on a single product (i.e., Mozilla Firefox). Finally, they lack practical implementation, which prevents their actual usage and further evaluation.

Several theoretical guidelines for approaching the problem of software security assessment have been also proposed. For instance, (Zafar et al., 2015) proposed a way for constructing hierarchical security assessment models based on the Dromey's quality model (Dromey, 1995). 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 of 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. Also, (Medeiros et al., 2017) proposed a general theoretical methodology for assessing the trustworthiness (including security) of cloud-based web services, based on the adoption of multi-criteria decision-making techniques. However, they provide only a theoretical conceptualization of their overall idea, without providing specific details about their proposed assessment approach (e.g., model structure, potential metrics, etc.).

Recently, probably due to the lack of well-accepted security metrics, several researchers have started examining the feasibility of quantifying software security indirectly through the notion of Technical Debt (TD)[6]. More specifically, guidelines on how the concept of TD can be extended to support software security have been provided (Rindell et al., 2019; Rindell & Holvitie, 2019), whereas ways for prioritizing security bugs as technical debt items (i.e., quality issues) have already been proposed (Izurieta et al., 2018; Izurieta & Prouty, 2019). In addition, (Siavvas et al., 2019b) provided preliminary empirical evidence for the relationship between TD and software security, indicating that TD may potentially be used as an indicator of software security issues. However, TD needs to overcome important challenges before becoming a reliable measure of software security, since existing TD models address security risks inadequately, or not at all (Rindell & Holvitie, 2019).

2.2 Comparison and proposed advances

From the above analysis, it is clear that current literature lacks an effective model for measuring the internal security risk of software products. This 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 (Sentilles et al., 2018; Mohammed et al., 2016). In fact,

[6] Technical Debt is a notion inspired by the financial debt that is widely used in the software industry for assessing software quality (Cunningham, 1993).

according to our analysis, although a small number of security models have been proposed (Lai, 2010; Alshammari et al., 2011; Xu et al., 2013; Colombo et al., 2012; Zafar et al., 2015; Medeiros et al., 2018; Dayanandan & Kalimuthu, 2018), none of them has managed to achieve sufficient level of *practicality* and *reliability*.

This observation is supported by several recently published surveys (Ansar et al., 2018; Sentilles et al., 2018; Morrison et al., 2018), which highlight the lack of a well-accepted metric, model, or technique for quantifying software security. According to their findings, existing solutions are not reliable, since they are based on unreliable information and they lack sufficient empirical evaluation, whereas they are also impractical since they lack actual implementation.

A qualitative comparison of existing models is considered valuable for the reader, in order to gain a better understanding of their strengths and weaknesses, as well as of the main contributions of the present work. To realize this comparison, a set of criteria (i.e., characteristics) need to be defined, which will act as the main axes of the comparison. To this end, we defined five criteria that we consider important for characterizing the practicality and reliability of a given model (and against which we would like the models to be compared), which are listed below:

– **High-level overview:** The model provides quantitative values that measure inherent security properties of software products. These quantitative values (i.e., scores) can reflect (measure) the overall internal software security and/or important security requirements, such as Confidentiality, Integrity, and Availability.
– **Operationalization:** The model is operationalized, i.e., implemented in the form of a tool, and not only a theoretical conceptualization.
– **Automation:** The model is fully automated, and therefore it can be used regularly during the overall development process, without significantly affecting the developers' workflows.
– **Standardization:** The model encapsulates formal concepts that are expressed by international standards (e.g., ISO/IEC 25010).
– **Objectivity:** The model provides sufficiently objective security assessments, considering minor (to no) subjective information (i.e., information defined by the authors and not by well-accepted sources). In fact, it avoids 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, evaluation of its impact on the model's objectivity is provided.

The first three criteria are related to the *practicality* of a given security model, whereas the latter two are related to its *reliability*.

The qualitative comparison of the existing models based on the aforementioned criteria is presented in Table 1. It should be noted that our approach was inspired by similar approaches that are used in the literature for comparing different models, techniques, or systems (e.g., (Heitlager et al., 2007) and (Li et al., 2018)).

As can be seen in Table 1, none of the existing models manages to encompass all of the defined characteristics. 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 (in fact, author judgments) and arbitrarily selected parameters, while they also lack sufficient empirical evaluation. Their evaluation is based on simple

Table 1 Comparison between existing security models and the proposed Security Assessment Model (SAM)

Requirements	[1]	[2]	[3]	[4]	[5]	[6]	[7]	SAM
High-level overview	✓	✓	✓	✓	✓	✓	✓	✓
Operationalization	×	×	×	×	×	×	×	✓
Automation	✓	✓	✓	×	×	×	✓	✓
Standardization	×	×	×	✓	✓	✓	✓	✓
Objectivity	×	×	×	×	×	×	×	✓

[1]: (Lai, 2010); [2]: (Alshammari, et al. 2011); [3]: (Medeiros et al., 2018); [4]: (Xu et al., 2013); [5]: (Colombo et al., 2012); [6]: (Zafar et al., 2015); [7]: (Dayanandan & Kalimuthu, 2018)

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 (Verendel, 2009; Sentilles et al., 2018; Morrison et al., 2018; Ansar at al., 2018).

As discussed in detail in the rest of the paper, the proposed security model manages to encompass all of the defined characteristics. In brief, the model is highly *practical*, since it is operationalized in the form of a standalone tool (Online, 2020) and is fully automated, allowing its regular application during the overall development cycle. It is also sufficiently *reliable* (compared to its counterparts), since it is based on information retrieved from commonly accepted sources (e.g., ISO/IEC 25010 and CWE), while it was also built and evaluated on a large volume of empirical data (i.e., 250 software applications, comprising approximately 20 million lines of code). It should be noted that, for reasons of completeness, the model considers minor subjective information (i.e., expert judgments), which was not found to influence the overall assessment, and which can be completely eliminated (see Section 5.5).

At this point, some clarifications with respect to the concepts of vulnerability detection, vulnerability prediction, and security assessment are considered necessary to familiarize the reader with the content of the present paper. These concepts are actually three individual research directions in the broader field of software security, which often complement each other.

In particular, vulnerability detection focuses on building techniques and mechanisms able to detect actual vulnerabilities that reside in software programs (McGraw, 2008; Felderer et al., 2016; Mohammed et al., 2016). Such techniques include static analysis, penetration testing, fuzzing, etc., each one demonstrating its strengths and weaknesses (Howard & Lipner, 2006; Felderer et al., 2016). The research in the area of security assessment focuses on building techniques and models that are able to evaluate the security level of a software product, normally based on the results produced by vulnerability detection mechanisms (Verendel, 2009; Basso et al., 2019). For instance, a software product can be considered relatively secure, if no vulnerabilities can be detected. Finally, the research in the field of vulnerability prediction focuses on predicting software artefacts that are likely to contain vulnerabilities (Jimenez et al., 2016; Shin et al., 2011; Siavvas et al., 2018a; Scandariato et al., 2014). This information is very useful for the developers and project managers, for prioritizing their testing and fortification efforts, as limited test resources can be allocated to high-risk areas. For instance, more extensive vulnerability detection mechanisms can be applied to these vulnerability-prone components, increasing the probability of identifying actual vulnerabilities (Jimenez et al., 2016; Shin et al., 2011; Siavvas et al., 2018a; Scandariato et al., 2014;

Chowdhury & Zulkernine, 2011; Zhang et al., 2019). Therefore, from the above description, it is eminent that, although these concepts are treated as independent areas of research, they actually complement each other, and they are usually combined.

3 Security assessment model

In this section, we describe in detail the proposed security assessment model. 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 3.2.2).

It should be noted that the model was constructed based chiefly on the features provided by the *Quality Assessment Tool Chain (QATCH)* (Siavvas, 2017a), 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 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 it will be shown later, the platform was slightly extended, in order to consider security-specific concepts, and, thus, 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.

3.1 Model description

3.1.1 General structure

The general structure of the proposed model is illustrated in Fig. 2. As it is shown in Fig. 2, 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 and uses their values along with a set of *thresholds*, in order to assign ratings (i.e., scores) to a group

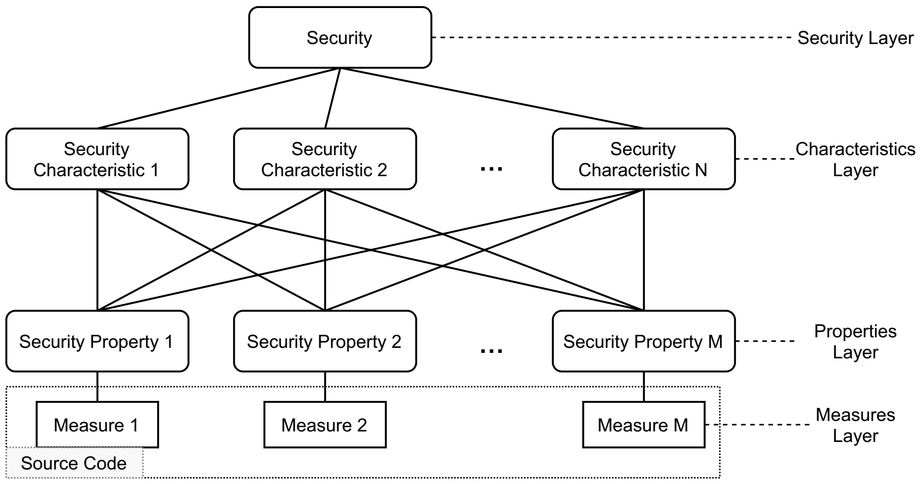


Fig. 2 The overall structure of the proposed security assessment model (SAM)

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. The score 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 product's security level to developers and project managers, compared to the individual low-level measures.

The overall structure of the proposed model is in line with the guidelines that are provided by the ISO/IEC 25010 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 (Wagner, 2013) 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 subjective beliefs. Similar attempts have been conducted for other quality attributes, like *Maintainability* (e.g., (Heitlager et al., 2007)).

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 (Saaty, 2008). This is the main reason why hierarchical decomposition is commonly used for decision making, as well as for the construction of both quality (Wagner et al., 2015) and security models (Alshammari et al., 2011; Xu et al., 2013; Colombo et al., 2012; Dayanandan & Kalimuthu, 2018). 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 thus enabling root-cause analysis (Heitlager et al., 2007).

3.1.2 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 2). The reasoning behind the selection of these characteristics is that they constitute fundamental and well-established security objectives (i.e., requirements) of software systems (Andress, 2014), recognized by numerous international standards as the core information security attributes, e.g., (NIST, 2018; ISO, 2013), which together form the CIA triad of information security (Andress, 2014). In fact, these three security attributes are considered the principal components for building secure systems (Whitman & Mattord, 2011; Mumtaz et al., 2018). Another reason for their selection is that they can be assessed based on information retrieved from the application source code through static analysis. For instance, most of the available static code analyzers are able to detect security issues that infringe the security attributes of *Confidentiality*, *Integrity*, and *Availability*, as can be seen by the entries of common weaknesses that are available on the Common Weakness Enumeration (CWE) knowledge base.

It should be noted that additional security attributes were considered for the construction of the model (with the purpose to complement the CIA triad), however, they were excluded either because they could not be assessed through static analysis (e.g., Authenticity), or because the derivation of a reliable set of weights was impossible (e.g., Non-repudiation), due to limited information available (see Section 3.2.2). Although this decision can be considered subjective, as it was deliberately made by the authors of the model, deciding to keep those security characteristics inside the model, would add more subjectivity and would affect the correctness of the produced model, since arbitrary decisions would have to be made regarding their quantification through static analysis and the selection of their weights. More specifically, we would need to state that specific characteristics, e.g., Authenticity, can be quantified through completely irrelevant alert-based properties, whereas for those characteristics that our weights elicitation approach could not produce a reliable set of weights (due to lack of reliable information on well-established knowledge

Table 2 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 widely known as the CIA triad (Andress, 2014). The definitions were retrieved from the ISO/IEC 25010 (ISO, 2011) 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 (ISO, 2011).
Integrity	The degree to which a software product, system, or component prevents unauthorized modification of sensitive data (ISO, 2011).
Availability	The degree to which a system, product, or component is operational and accessible when required for use (ISO, 2011).

bases), we would need to assign weights based on our intuition and expertise. This would infringe the reliability and the correctness of the produced model. After all, as already mentioned, the selected security characteristics (i.e., *Confidentiality*, *Integrity*, and *Availability*) are considered in the literature as the main security requirements of information systems in general (NIST, 2018; ISO, 2013), which are often termed as the CIA triad (Andress, 2014). Hence, focusing on characteristics for which we are confident that there is reliable information expressed by well-accepted sources, is better than extending our model for covering more cases, with the risk of infringing its correctness and its reliability.

The model also consists of 11 security properties. These properties are evaluated (i.e., quantified) 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 in Fig. 2, 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) alert-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 & Kemerer, 1994) and QMOOD (Bansiya & Davis, 2002), and calculated through the CKJM Extended[7] software metrics tool, which is integrated in the QATCH (Siavvas, 2017a) platform. CKJM Extended is an open source tool able to calculate a wide range of metrics (including those proposed by (Chidamber & Kemerer, 1994) and (Bansiya & Davis, 2002)), by processing Java files. The aforementioned metrics-based properties and their corresponding metrics are described in Table 3.

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. In particular, a large number of empirical studies have shown that software metrics are related to the existence of software vulnerabilities in a statistically significant manner (albeit with weak strength) and that they are capable of accurately predicting the existence of vulnerabilities in software components (e.g., (Shin & Williams, 2008; Chowdhury & Zulkernine, 2010; Chowdhury & Zulkernine, 2011; Shin et al., 2011; Moshtari et al., 2013; Moshtari & Sami, 2016)). However, instead of being based on the generalizability of these results, in a recent empirical study (Siavvas et al., 2017b), which was based on the code base that was used for the calibration of the present model (see Section 3.2.1), we verified the capacity of the software metrics to indicate the existence of security issues in the selected products. This study provides strong evidence for the inclusion of software metrics in the model.

As illustrated in Table 3, each metric-based property (i.e., metric type) is measured by a single software metric. The reasoning behind our decision to assign one metric as the representative metric for the quantification of each metric-based property was to avoid making the produced security model complex. The selection of the representative metrics was based mainly on evidence that could be found in the related literature. In fact, the CKJM Extended tool offers a variety of software metrics for each one of the selected metric types. During the construction of the security model, we examined all the metrics that the CKJM Extended tool offers, and for each metric type (i.e., metric-based property) we chose the

[7] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

Table 3 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 & Kemerer, 1994) and (Bansiya & Davis, 2002) metric suits (also known as CK and QMOOD suits respectively), while they are quantified using the CKJM Extended tool

Property	Description	Metric
Complexity	The level of the logical complexity of the software product.	*WMC (Weighted Methods per Class)*: 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.	*LCOM (Lack of Cohesion in Methods)*: The number of methods 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.	*CBO (Coupling Between Objects)*: The total number of classes coupled to a given class.
Encapsulation	The degree to which the software product avoids user-defined information.	*MOA (Measure of Aggregation)*: This metric is a count of the number of data declarations that are user-defined.

one that we could find significant evidence in the literature regarding its ability to indicate the existence of vulnerabilities in software. In fact, sufficient evidence exists in the related literature for the relationship of WMC, CBO, and LCOM to software security (i.e., the existence of vulnerabilities) (Shin & Williams, 2008b; Chowdhury & Zulkernine, 2011; Shin et al., 2011; Medeiros et al., 2017; Moshtari et al., 2013; Moshtari & Sami, 2016; Stuckman et al., 2017; Ferenc et al., 2019; Jimenez et al., 2019; Zhang et al., 2019), whereas a previous empirical study that we conducted on the same dataset that was used for the calibration of the proposed security model revealed that MOA is the encapsulation metric that demonstrates the closest correlation to software security (at least for the given dataset) (Siavvas et al., 2017b). Since the rest of the metrics that the CKJM Extended tool offers have not been studied (or extensively studied) in the literature for their relevance to software security, we decided not to include them in the model, in order to prevent putting under question the correctness and reliability of the proposed model. It should be noted that, similarly to (Baggen et al., 2012), our final selection was also influenced by the variability of the values of the different metrics that were calculated based on the selected benchmark repository (see Section 3.2.1). In fact, metrics with very small variability were excluded, as they would probably lead to unreliable set of thresholds (see Section 3.2.1).

The model also contains 7 alert-based properties, namely *Null Pointer*, *Assignment*, *Exception Handling*, *Resource Handling*, *Logging*, *Misused Functionality*, and *Synchronization*. Each one of these groups contains weaknesses that may lead to similar vulnerabilities, and therefore they are also termed as vulnerability categories.

These vulnerability categories were constructed by properly grouping the security-related rules of the PMD[8] static code analyzer. PMD, which is also integrated in the QATCH framework, is an open-source static code analyzer that is included by both OWASP[9] and NIST[10] in their lists of recommended static analysis tools that can be used

[8] https://pmd.github.io/

[9] https://www.owasp.org/

[10] https://www.nist.gov/

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 found, an alert is produced, which is added in a broader evaluation report.

Despite its ability to detect issues with security implications, PMD does not provide any ready-made security-specific groups of rulesets, nor information regarding the security relevance of its rules. For this purpose, we manually identified the security-relevant rules and grouped them into vulnerability categories. To ensure correct definition of the vulnerability categories we used information from the *Common Weakness Enumeration (CWE)*, which is a commonly 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 order to facilitate the construction of the vulnerability categories. In fact, the initial set of 233 rules for Java programming language, which are provided by the PMD version that is used by the QATCH platform, were inspected with respect to their security relevance. For each one of these rules, we manually searched in the CWE database to find a relevant entry. In Table 4 we provide some representative examples of our mapping. It should be noted that our mapping is in line with the one provided by CodeSonar[11], while we also cover some additional rules. Those rules that could not be mapped to a security-relevant CWE entry were excluded from the final set of rules. This process led us to a set of 138 PMD rules with security implications, which were grouped into the aforementioned seven vulnerability categories (i.e., alert-based properties), based on their relevance. A description of these categories is provided in Table 5, along with the most representative CWE ID that better describes each group.

The detailed list of PMD rules that belong to each vulnerability category, along with their mapping to the CWE entries is provided at (Online, 2020), which includes supporting material of this paper. This mapping was also required for the derivation of the model's weights through a novel approach that is described in Section 3.2.2. A similar approach for defining both quality- and security-related weakness categories was adopted in a recent empirical study (Siavvas et al., 2017b).

At this point, it should be noted that the reasoning behind the selection of a single static code analyzer for quantifying the alert-based properties is that we wanted the produced security model to be less complex and highly practical. Although integrating multiple static code analyzers would allow the model to detect additional types of security issues, it would also lead to a significant increase in the information (both actionable and unactionable) reported to the user. Actually, despite the fact that static analysis tools are known for

Table 4 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()
NullAssignement	CWE-476: NULL Pointer Dereference

[11] https://www.grammatech.com/products/codesonar

Table 5 The alert-based properties (i.e., vulnerability categories) of the proposed security assessment model. The most representative CWE entry that better describes each property is also provided

Vulnerability Category	Description	CWE
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-668: Exposure of Resource to Wrong Sphere
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.	CWE-388: Error Handling, CWE-199: Information Manag. Errors
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.	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

their effectiveness in detecting vulnerabilities early enough in the software development lifecycle (SDLC) (McGraw, 2006; Chess & McGraw, 2004; Mohammed et al., 2016), they are underused in practice (Johnson et al., 2013; Bholanath, 2016). The main reason for their limited adoption is the overwhelming information that they produce, which are long lists of raw warnings (i.e., alerts) (Johnson et al., 2013). These warnings need to be manually examined to determine whether they correspond to actual bugs that require immediate corrective actions (i.e., actionable alerts (Heckman & Williams, 2009)) or they are just false positives (i.e., unactionable alerts). This process is normally called triaging (Walden & Doyle, 2012), and it is a highly time-consuming and effort-demanding process. The large number of unactionable alerts (i.e., false positives), discourages developers from using them in practice. Therefore, aggregating multiple static code analyzers, would also aggregate their reports, i.e., both their actual issues and the false positives, discouraging developers from using the proposed model in practice. Of course, the reader, following the approach presented in this paper, can build similar models, utilizing multiple static code analyzers, if deemed necessary.

It should be noted that, apart from reasons of model automation, since the proposed SAM focuses on the coding phase of the overall SDLC, static analysis was the most suitable mechanism on which it should be based for assessing code-level security. As already mentioned, most of the vulnerabilities that reside in software stem from a small number of common programming errors that are introduced by the developers during the coding (i.e., implementation) phase of the overall SDLC (Krsul, 1998; Howard et al., 2010; Chess & McGraw, 2004; Mohammed et al., 2016). For instance, well-known security breaches like Equifax Breach (Luszcz, 2018) and Heartbleed (Carvalho et al., 2014) were caused by simple implementation errors. Static analysis is considered one of the most effective techniques for detecting such code-level security-related errors, early enough in the overall SDLC, when their correction is relatively cheap and easy (Chess and McGraw, 2004; McGraw, 2006; Mohammed et al., 2016), and therefore it should be applied during the coding phase of the overall software development. This belief is expressed by several experts in the field of software security (e.g., (Chess & McGraw, 2004; Chess & McGraw, 2004)), while almost all the well-established secure software development lifecycles (SSDLCs), including the well-known Microsoft's SDL (Howard, 2003; Howard & Lipner, 2006), OWASP's OpenSAAM[12], and Cigital's Touchpoints (McGraw, 2006), 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, static analysis is a security activity commonly adopted by major technology firms like Google, Microsoft, Adobe and Intel, as reported by the BSIMM[13] initiative.

However, one of the main shortcomings of static analysis is that it produces false positives, i.e., alerts that do not correspond to actual issues (Johnson et al., 2013). Although the false positives produced by static analysis is not expected to affect the overall assessment performed by the model, as it is based on the concept of benchmarking, it can affect its practicality. In brief, they could lead to large lists of alerts that need to be examined by the developers in order to verify which of them are actionable and require to be fixed and which do not. This process, which is known as triaging (Walden & Doyle, 2012; Ruthruff, 2008), is highly time consuming and effort demanding, and is known to be the main reason that static analysis tools are underused in practice (Johnson et al., 2013). Hence, since the produced model is based on static analysis, the practicality of static analysis actually affects the practicality of the produced model.

Hence, similarly to (Walden et al., 2009), we decided to examine the false positive rate of the selected static code analyzer (i.e., the PMD tool), in order to ensure that it is bounded within manageable levels. However, the number of static analysis alerts that were produced for each one of the software products of the selected benchmark, as well as the fact that these products were not developed by us, made it impractical to manually verify whether each vulnerability was false positive or not, especially in a reliable way. Instead, similarly to (Walden et al., 2009), we examined two individual projects. In particular, in order to ensure the correctness of the analysis and the reliability of the observations, we selected two software projects that were developed within the context of the SDK4ED EU Project, for which their developers were reachable. For these two software projects, the PMD tool reported 149 issues, 23 of them were false positives. Hence, the estimated false

[12] https://www.opensamm.org/

[13] https://www.bsimm.com/

positive rate for the PMD analyzer was found to be 15.4%. According to our opinion, this false positive rate can be considered tolerable, and it is not expected to affect the practicality of our proposed SAM.

3.1.3 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 low security, while values closer to 1 indicate better level of security.

Similarly to other quality models (e.g., (Wagner et al., 2015; Heitlager et al., 2007; Siavvas et al., 2017b)), 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 model's characteristics from the properties ratings. 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 what follows.

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 7 vulnerability categories defined in Table 5. It also calculates the absolute values of the four software metrics defined in Table 3. These measures cannot be used directly for the evaluation of the model properties, since they are in a raw and low-level format. More specifically, a list of static analysis alerts is produced for each one of the 7 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., 2012) 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, such as those proposed by (Deissenboeck et al., 2009) and (Wagner et al., 2015).

As far as the vulnerability categories (i.e., alert-based properties) are concerned, QATCH has been extended, in order to support the calculation of the *static analysis-vulnerability density metric (SAVD)* (Walden and Doyle, 2012; Walden et al., 2009). As mentioned in Section 2.1, 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:

$$SAVD_i = 1000\frac{N_i}{LOC} \tag{1}$$

where:

$SAVD_i$: the $SAVD$ of the ith 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. We also used a similar approach in another empirical study (Siavvas et al., 2017b). 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 Fig. 2). 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 by (Wagner et al., 2012), 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:

$$f_p(s) = \begin{cases} 1 & , \ s \le t_l \\ \dfrac{0.5}{t_l - t_m}(s + t_l - 2t_m) & , \ t_l \le s \le t_m \\ \dfrac{0.5}{t_m - t_u}(s - t_u) & , \ t_m \le s \le t_u \\ 0 & , \ s \ge t_u \end{cases} \tag{2}$$

In the above formula, s corresponds to the system-level (i.e., normalized) value of the code-level measure that is associated to the corresponding property P. For better understanding, the main structure of the *utility function* is presented in Fig. 3. It should be noted that the main reasoning behind the selection of this structure of *utility function*is that it has been also used by well-known quality models (e.g., (Wagner et al., 2015; Wagner et al., 2012)).

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 in Fig. 2, 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 3.2.2, 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

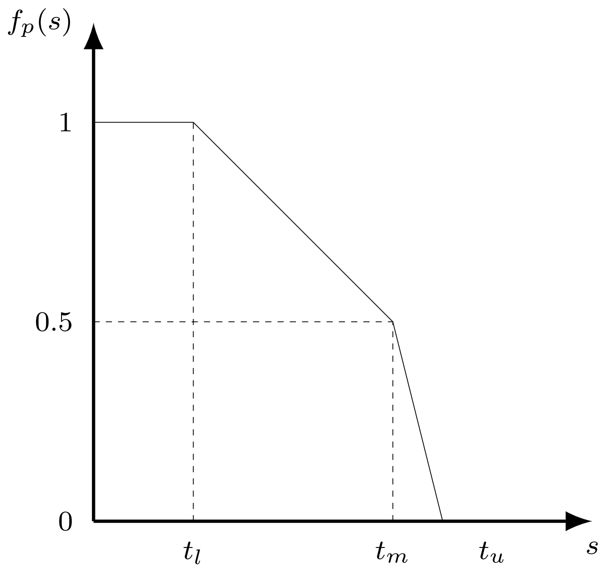


Fig. 3 The general structure of the utility functions of a given security property *P*. The utility function assigns a continuous value between 0 and 1 to the corresponding property, based on the system-level (i.e., normalized) value *s* of its associated code-level measure and on a set of data-driven thresholds (i.e., t_l, t_m, t_u)

(Andress, 2014). 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. However, it should be noted that the model is highly customizable with respect to its weights and thresholds, and therefore the importance of its security characteristics on the overall *Security Index* can be manually defined by the users based on their preferences. This can be easily achieved simply by changing the values of their corresponding weights in the XML file that contains the description of the model (available at (Online, 2020)), without the need for any further configuration or re-calibration.

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 perceive better 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 6. 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 (Heitlager et al., 2007)). In Table 6, we also report the percentage of the software products of the code base that reside in each rating group. From Table 6, 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.

Finally, it should be noted that the produced *Security Index* is a relative (and not absolute) security score. This means that the *Security Index* actually denotes how well the analyzed software product stands with respect to its security compared to the benchmark (i.e., population) of well-known software products. For instance, if a software product receives 5 stars (i.e., a *Security Index* above 0.8), this indicates that the analyzed software is

Table 6 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%

comparable to the top 10% of the software products of the selected benchmark with respect to their security level. Hence, this information provides insight for the security readiness of a given software product, allowing project managers to decide whether it is ready to be released to the market. The *Security Index* is a relative score since the overall assessment that is performed by the proposed SAM is based on the popular idea of benchmarking. Being based on benchmarking, also allows the model not to be affected (at least significantly) by the errors of the applied tools on which the assessment is based, as the same tools are applied both on the benchmark and on the newly analyzed software, and therefore these errors are neutralized during the comparisons, assuming (without loss of generality) that the selected tools are not biased.

3.2 Model construction

As can be seen in Fig. 1, one of the main steps of our work is the model construction. From the previous analysis, it is clear that the proposed SAM 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 international standard, (ii) it encapsulates well-established concepts of quality models (e.g., aggregation (Heitlager et al., 2007), utility functions (Wagner et al., 2012), and (iii) it is based on low-level measures with proven 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 3.1, 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. Hence, this section presents the details of how the proposed security assessment model was constructed, and particularly how the model's *thresholds* and *weights* were derived.

3.2.1 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 proprieties, 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., (Heitlager et al., 2007; Wagner et al., 2012; Siavvas et al., 2017a; Vale et al., 2019)), 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 100 most popular Java applications (including well-known libraries like Junit, Xerces, HyperSQL, etc.), which were able to be analyzed with the selected tools (see Section 3.1), were retrieved from the Maven Repository, which, as already stated, is the largest online repository of Java libraries. The final repository comprises 22,259 classes (i.e., source code files), which correspond approximately to 7 million lines of code (LOC). As already mentioned, the same code base was used in a relatively recent empirical study, with the purpose to investigate the ability of software metrics to indicate the existence of vulnerabilities in software products (Siavvas et al., 2017b).

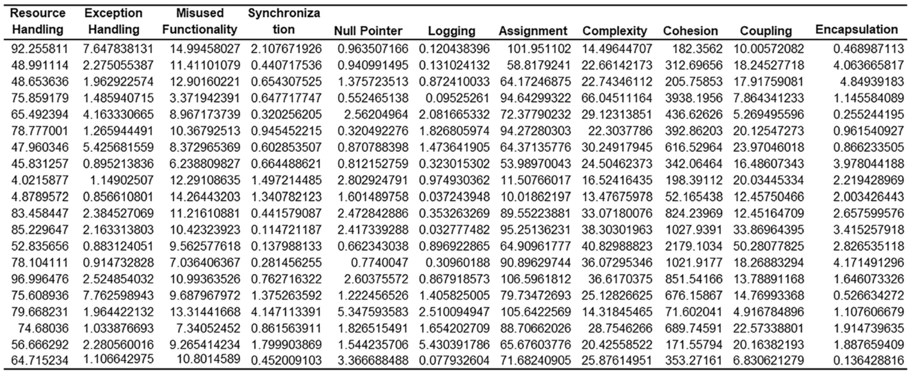
It should be noted that by being based on the Maven Repository we ensure that the final benchmark comprises popular software applications that are available on the market and that are widely used in practice, avoiding applications produced by unreliable parties (e.g., students) or applications that their usage or development has been abandoned. The repository includes software projects of varying size, ranging from very small software applications (i.e., 1,055 LOC) to really large applications (i.e., 521,262 LOC). The average size of an application that belongs to the benchmark repository is 63,396 LOC. Finally, it should be noted that the final benchmark repository contains products from different domains, including Web Development, Multimedia, Software Verification and Validation, Data management, Communication, Security, DevOps, Cloud Computing, and Distributed Computing. All these provide confidence that the resulting benchmark repository is a representative population of software applications.

Initially, QATCH was adopted 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 5 and Table 3). 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 broad repository is illustrated in Fig. 4.

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 (Wagner et al., 2015) were used for deriving these thresholds:

$$\begin{aligned} t_l &= \min(\{s : s \geq Q_{25\%}(s_1, ..., s_n) - 1.5 \cdot IRQ(s_1, ..., s_n)\}) \\ t_m &= \text{median}(s_1, ..., s_n) \\ t_u &= \max(\{s : s \leq Q_{75\%}(s_1, ..., s_n) + 1.5 \cdot IRQ(s_1, ..., s_n)\}) \end{aligned} \tag{3}$$

where:



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

Fig. 4 The system-level values of the security properties of 20 software applications retrieved from the complete benchmark repository used for the calibration of the model

s_i denotes the normalized value of the measure s of the i-th benchmark product
Q_p denotes the p-percentile
$IRQ(s_1, ..., s_n)$ denotes the inter-quartile-range

In simple words, after performing outlier removal, 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 7. For reasons of reproducibility, the full list of the calculated measures, along with the script that calculates the thresholds based on Equations (3) are available on the website with the supporting material of the present study (Online, 2020).

3.2.2 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 subjective judgments of individual experts, which are typically the authors of the models in the related literature. However, being based on the opinions of a limited number of individual experts is known

Table 7 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

P1 Resource Handling; *P2* Exception Handling; *P3* Misused Functionality; *P4* Synchronization; *P5* Null Pointer; *P6* Logging; *P7* Assignment; *P8* Complexity; *P9* Cohesion; *P10* Coupling; *P11* Encapsulation to affect the reliability (i.e., trustworthiness) of the proposed models, due to the fact that the performed assessment may not be sufficiently objective.

Hence, as opposed to previous attempts, in our work we propose a novel approach that allows the derivation of weights that reflect the well-established knowledge expressed by CWE, replacing, in that way, subjective judgments expressed by a limited number of individuals based on their expertise (i.e., 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) (Saaty, 2008) and the SMARTS/SMARTER (Edwards & Barron, 1994) approach, for deriving a reliable set of weights that reflect the information provided by CWE.

The Analytic Hierarchy Process (AHP) (Saaty, 2008) 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. Multi-criteria decision-making techniques are considered valuable for performing security assessment (Medeiros et al., 2018), as it is a multi-dimensional quality attribute (ISO, 2011).

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 8.

As can be seen in Table 8, 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:

$$A = (e_{ij}), \quad where : (i,j = 1, 2, ..., M) \tag{4}$$

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

$$(A - \lambda I) = 0 \tag{5}$$

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

$$\bar{w} = \frac{\bar{w}_{\max}}{\sum_{i=1}^{M} w_{\max,i}} \tag{6}$$

where:

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

Table 8 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	...	P_j	...	P_M
P_1	e_{11}	e_{12}	...	e_{1j}	...	e_{1M}
P_2	e_{21}	e_{22}	...	e_{2j}	...	e_{2M}
$\vdots$	$\vdots$	$\vdots$		$\vdots$		$\vdots$
P_i	e_{i1}	e_{i2}	$\cdots$		...	e_{iM}
				e_{ij}		
$\vdots$	$\vdots$	$\vdots$		$\vdots$		$\vdots$
P_M	e_{M1}	e_{M2}	$\cdots$	e_{Mj}	...	e_{MM}

– $w_{max,i}$: the ith entry of the principal eigenvector
– M : the length of the principal eigenvector

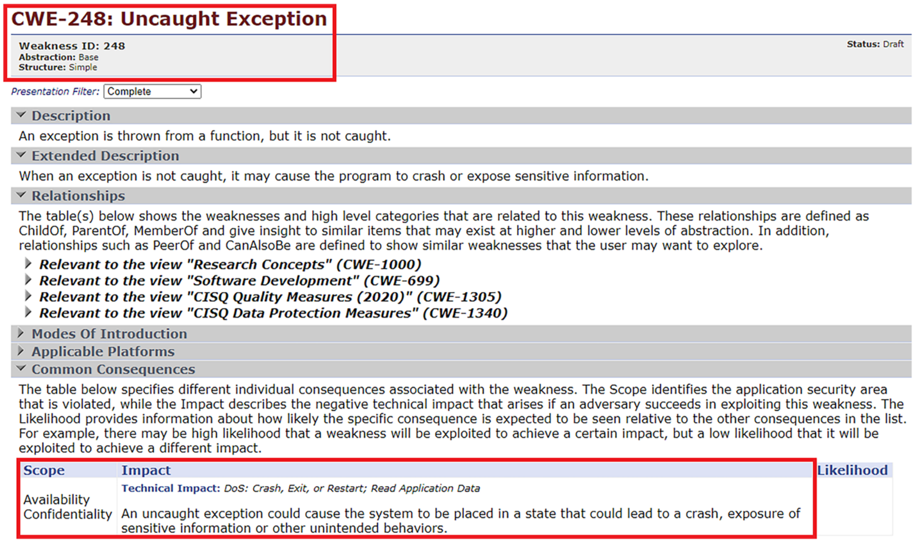
The main advantage of AHP is that the experts have to focus only on a single pair of properties at each time, while the interdependencies between the properties are automatically considered during the weights calculation. This approach guarantees that the produced weights will reflect the expert judgments (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 commonly accepted sources of information, instead of individual expert judgments.

As far as the alert-based properties are concerned, the CWE knowledge base is used as the source of information. In particular, as described in Section 3.1.2, 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 in Fig. 5, 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 alert-based properties (i.e., vulnerability categories) on the security characteristics of the model.

Initially, for each one of the alert-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 model's 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:

$$I_{E,C} = impact(E, C) = \begin{cases} 1, & \text{if } E \text{ affects } C \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

Based on the identified binary impacts of the individual CWE entries, the impacts of the alert-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



CWE-248: Uncaught Exception

Weakness ID: 248
Abstraction: Base
Structure: Simple

Status: Draft

Presentation Filter: Complete

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 "Software Development" (CWE-699)
- Relevant to the view "CISQ Quality Measures (2020)" (CWE-1305)
- Relevant to the view "CISQ Data Protection Measures" (CWE-1340)

Modes Of Introduction

Applicable Platforms

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 Confidentiality	Technical Impact: DoS: Crash, Exit, or Restart; Read Application Data 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.	

Fig. 5 CWE provides information about the impact that each weakness has on the main security requirements of software products

a reported impact on each one of the N characteristics of the model, i.e., $C_1, C_2 \ldots, C_N$. 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:

$$I_{P_i,C_k} = impact(P_i, C_k) = \frac{N_{P_i,C_k}}{\sum_{k=1}^{N} N_{P_i,C_k}} = \frac{\sum_{j=1}^{L} I_{E_{i,j},C_k}}{\sum_{k=1}^{N}\sum_{j=1}^{L} I_{E_{i,j},C_k}} = \frac{\sum_{j=1}^{L} impact(E_{i,j}, C_k)}{\sum_{k=1}^{N}\sum_{j=1}^{L} impact(E_{i,j}, C_k)} \tag{8}$$

where:

- N_{P_i,C_k} : the number of the CWE entries of P_i (i.e., $E_{i,1}, E_{i,2}, \ldots, E_{i,L}$) 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), given by Equation (7).
- M : the number of properties of the model.

In simple words, from Equation (8) it is shown that 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 (7).

Based on the aforementioned approach we calculated the impact scores of all the alert-based properties of the proposed model (see Table 9), on all the characteristics of the model, namely *Confidentiality*, *Integrity*, *Availability*. It should be noted that the characteristic of *Non-repudiation* was also considered in the initial analysis. The calculated impact scores are presented in Table 9, due to the fact that relevant information was available. 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.

The impact scores presented in Table 9 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, as 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 reliable 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 (Edwards & Barron, 1994) 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 9). A fixed relative impact score r_{ik} is assigned to property P_i, based on the following equation:

$$r_{ik} = f_{SMARTS/SMARTER}(M, rank_{ik}) \tag{9}$$

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

Table 9 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

their position in the ranked list. In Table 10, the weights that are assigned by the SMARTS/SMARTER approach for a varying number of properties are shown.

To calculate the relative impact scores, each column of Table 9 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 11 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. The inclusion of this attribute would require the properties to be ranked manually based on our expertise. However, this would infringe the objectivity, and, in turn, the reliability of the produced model, so its exclusion was considered the best approach. From Table 11, it is obvious that the assigned relative impact scores are the weights of Table 10 for $M = 7$.

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., alert-based properties). A simple procedure for completing these cells of the PWC matrices based on the relative scores of Table 11 was followed. Suppose the PWC matrix of the characteristic C_k presented in Table 8, 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 11). The expert judgment e_{ij} is calculated as follows:

$$e_{ij} = \begin{cases} \left\lceil \frac{(r_{ik}/r_{jk})}{2} \right\rceil & \text{if } r_{ik} > r_{jk} \\ \left\lceil \frac{(r_{jk}/r_{ik})}{2} \right\rceil & \text{if } r_{ik} < r_{jk} \end{cases} \tag{10}$$

The above formula ensures that the final judgments will lie in the interval [0.1, 9], as suggested by (Saaty, 2008).

Table 10 The weights assigned by the SMARTS/SMARTER approach for a various number of elements (i.e., *M*). The table is adapted from (Edwards & Barron, 1994)

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

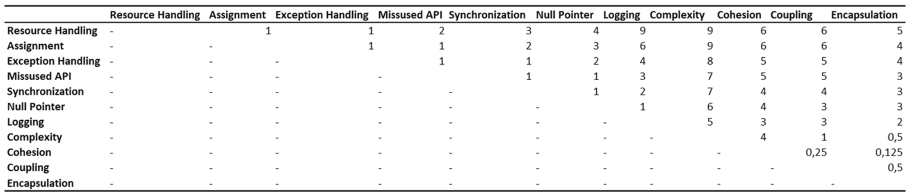
Table 11 The rankings of the model's properties for each characteristic, along with their relative impact scores, retrieved from the SMARTS/SMARTER (Edwards & Barron, 1994) approach

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

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 commonly 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 model's weights were calculated based on Equations (5) and (6). As an example, the PWC matrix that was used for the calculation of the weights of *Confidentiality* is illustrated in Fig. 6. The other two PWC matrices are available on the website with the supporting material of the present work (Online, 2020). The final weights of the model's characteristics are presented in Table 12. The final weights, seem to reflect both the knowledge retrieved by CWE knowledge base (i.e., the rankings presented in Table 11), and the judgments provided by the experts.

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)[14] (Shin & Williams, 2008b; Chowdhury & Zulkernine, 2011; Shin et al., 2011; Medeiros et al., 2017; Siavvas et al., 2017b; Moshtari et al., 2013;



	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	-	-	-	-	-	-	-	-	-	-	-

Fig. 6 The pairwise comparison matrix used for the calculation of the model's weights that correspond to the characteristic of Confidentiality

[14] A large volume of research endeavors have shown that software metrics are related to software vulnerabilities in a statistically significant manner, albeit with weak strength (Shin & Williams, 2008b; Chowdhury & Zukernine, 2011; Shin et al., 2011; Medeiros et al., 2017; Siavvas et al., 2017b; Moshtari et al., 2013; Moshtari & Sami, 2016; Stuckman et al., 2017; Ferenc et al., 2019; Jimenez et al., 2019; Zhang et al., 2019).

Table 12 The final weights of the model's characteristics as derived from the AHP (Saaty, 2008) 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

Moshtari & Sami, 2016; Stuckman et al., 2017; Ferenc et al., 2019; Jimenez et al., 2019; Zhang et al., 2019), 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 considered stronger indicators of software vulnerabilities compared to software metrics, higher weights were assigned to the alert-based properties than to metric-based properties.

In fact, both static analysis alerts and software metrics are considered vulnerability indicators by the security model, as they indicate the existence of vulnerabilities in software components. However, the static analysis alerts are considered stronger indicators, since they actually locate the potential vulnerability, contrary to the software metrics that do not provide any information about the exact location (i.e., exact line of code) or even the type of the vulnerability that potentially exists in the associated software component. In addition to this, as already stated, software metrics have been found to be related to the existence of vulnerabilities in a statistically significant manner, albeit with a weak strength (Shin & Williams, 2008b; Chowdhury & Zulkernine, 2011; Shin et al., 2011; Medeiros et al., 2017; Siavvas et al., 2017b; Moshtari et al., 2013; Moshtari & Sami, 2016; Stuckman et al., 2017; Ferenc et al., 2019; Jimenez et al., 2019; Zhang et al., 2019). With that being said, smaller weights have been assigned to metric-based properties. In particular, as can be seen in Table 12, 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. 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 5.5). 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.

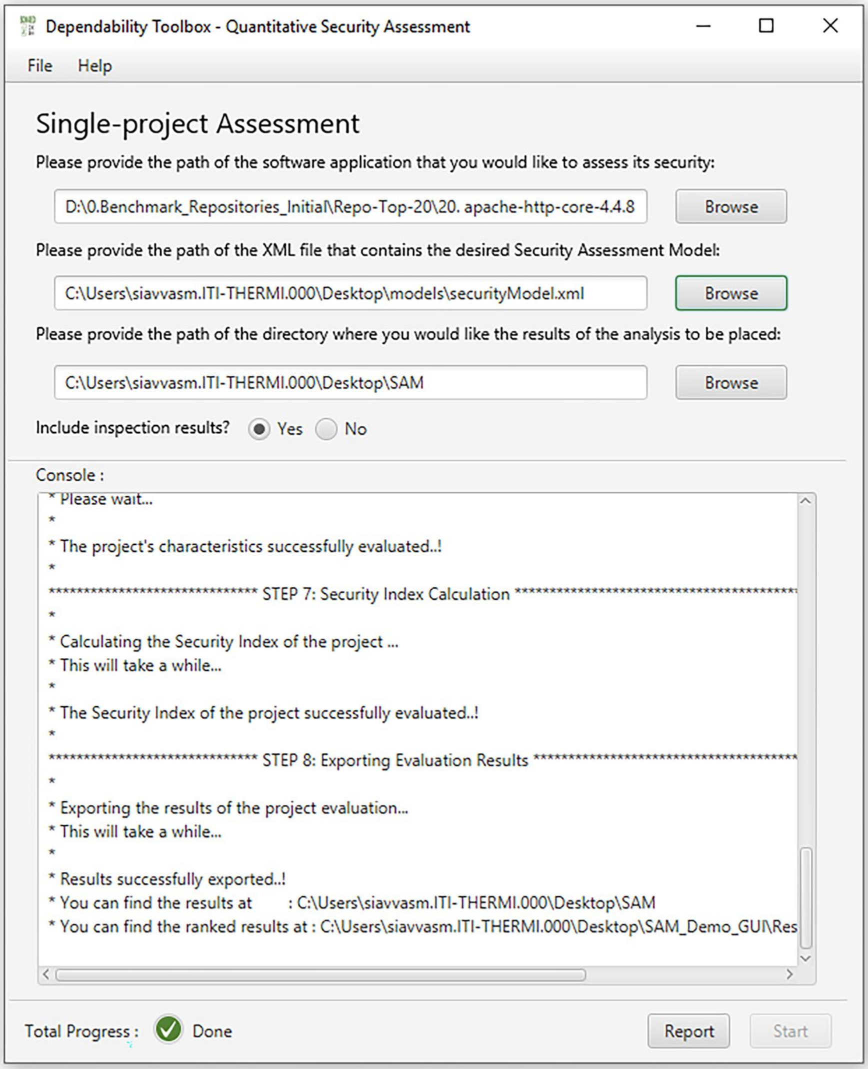
To sum up, the weight elicitation approach presented in this section can be used in practice for the derivation of a reliable set of weights, which are free (to the highest possible extent) from the subjective information that is incurred by expert judgments (e.g., the judgments of the authors of the produced model). 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 (Heitlager et al., 2007; Wagner et al., 2012), and therefore limits their adoption in practice.

4 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 in this gap, 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 at (Online, 2020), along with detailed instructions for its use. The command line version of the model allows its integration with existing continuous integration and assessment platforms, whereas 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 alternative version of the tool that offers a simple graphic user interface (GUI) is also provided. The main screen of this simple GUI is illustrated in Fig. 7.

The proposed SAM has been developed within the context of the SDK4ED 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 (with emphasis on Embedded Systems), 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 has been also developed as a web service, which has been integrated in the final SDK4ED platform, in order to be used as the main indicator of *Software Security*. Figure 8 depicts the current status of the Security Dashboard of the SDK4ED platform, which is based on the results produced by the proposed SAM.

As already mentioned, a similar SAM for evaluating software products written in C/C++ was also built within the context of the SDK4ED Project, following the steps that were followed for constructing the SAM for Java that were presented in Section 3. A detailed description of the internal structure and characteristics of this model is provided in the Appendix section of the paper. Similarly to the SAM for Java, the SAM for C/C++ has been implemented as a standalone command-line tool that can be found on the web page with the supporting material of the present work (Online, 2020). In addition to this, it is integrated into the SDK4ED Platform, in the form of a web service.



Single-project Assessment

Please provide the path of the software application that you would like to assess its security:

D:\0.Benchmark_Repositories_Initial\Repo-Top-20\20. apache-http-core-4.4.8

Please provide the path of the XML file that contains the desired Security Assessment Model:

C:\Users\siavvasm.ITI-THERMI.000\Desktop\models\securityModel.xml

Please provide the path of the directory where you would like the results of the analysis to be placed:

C:\Users\siavvasm.ITI-THERMI.000\Desktop\SAM

Include inspection results? [✓ Yes] No

```
* Please wait...
*
* The project's characteristics successfully evaluated..!
*
***************************** STEP 7: Security Index Calculation *****************************************
*
* Calculating the Security Index of the project ...
* This will take a while...
*
* The Security Index of the project successfully evaluated..!
*
***************************** STEP 8: Exporting Evaluation Results **************************************
*
* Exporting the results of the project evaluation...
* This will take a while...
*
* Results successfully exported..!
* You can find the results at      : C:\Users\siavvasm.ITI-THERMI.000\Desktop\SAM
* You can find the ranked results at : C:\Users\siavvasm.ITI-THERMI.000\Desktop\SAM_Demo_GUI\Res
```

Total Progress : ✓ Done

Fig. 7 The graphic user interface of the offline version of the proposed security assessment model

At this point, a description of some technical details of the SDK4ED Platform are considered necessary, in order to facilitate the adoption of its security assessment features in practice. The SDK4ED Platform has been implemented following the Microservices Architecture Pattern (Wolff, 2016), which means that its main components have been implemented as individual microservices, which offer their features in the form of independent web services. Hence, the security assessment models for Java and C/C++ that were developed are part of the *Quantitative Security Assessment (QSA)* service of the SDK4ED Platform. This service can be deployed locally using the Docker Engine[15], and

[15] https://www.docker.com/

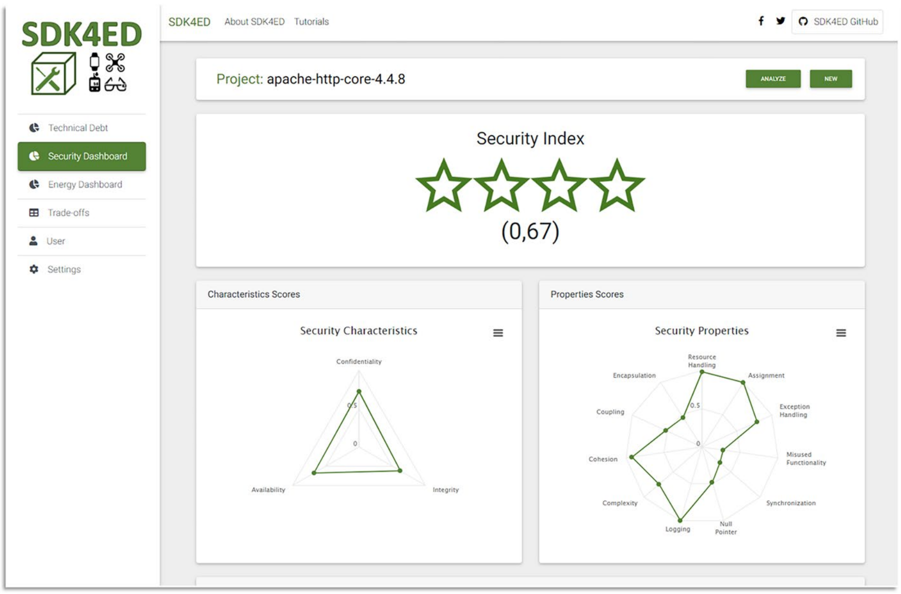


Fig. 8 The Security Dashboard of the SDK4ED platform

used independently for assessing the internal security level of software products. A Wiki page describing how this service can be installed and used for assessing the security level of a software product, either using the SAM for Java or the SAM for C/C++ has been created and made publicly accessible[16]. Finally, although this is the best way of utilizing the produced models, as it enables automation and integration with third-party software, instead of using this web service, one can use these models through the SDK4ED Dashboard. Information and tutorials on how the SDK4ED Dashboard can be installed locally and used for analyzing software products with respect to their *Maintainability*, *Energy Consumption*, and *Security* have been also provided online[17]. More information about the SDK4ED Platform can be found in our relevant publications (Jankovic et al., 2019; Siavvas et al., 2020b; Kehagias et al., 2021).

5 Model evaluation and discussion

As can be seen in Fig. 1, the second part of our work is the evaluation of the proposed model, in order to investigate its ability to provide reliable security assessments both at product and at class levels of granularity. As opposed to previous research endeavors (Ansar et al., 2018; Sentilles et al., 2018; Morrison et al., 2018), which lack empirical evaluation of their models, or they rely their evaluation on a limited number of simple case studies, in this paper we put significant emphasis on the elaborate empirical evaluation of our model, as it is highly required for ensuring its reliability. More specifically, in order to evaluate the correctness of the proposed SAM, as well as to identify its strengths

[16] https://gitlab.seis.iti.gr/sdk4ed-wiki/wiki-home/wikis/dependability-toolbox
[17] https://gitlab.seis.iti.gr/sdk4ed-wiki/wiki-home/wikis/home

and weaknesses, five 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 what follows.

5.1 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 commonly accepted and operationalized security assessment model exists in the literature, while, as opposed to software quality (Wagner et al., 2012; Siavvas et al., 2017a; Wagner et al., 2015), no expert-based security evaluations of software products are publicly available, in order to be used as reference point for 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)* (Walden et al., 2009), calculated at system level.

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). For the quantification of the *SAVD*, the FindBugs static code analyzer (Hovemeyer & Pugh, 2004) was employed, which is a popular open-source static analysis tool for Java, used in the literature for security auditing purposes (e.g., (Goseva-Popstojanova & Perhinschi, 2015; Siavvas et al., 2018b)). 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 FindSecurityBugs[18], which is a popular FindBugs plugin.

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 successfully compile with Maven without any errors or warnings. This process resulted to a relatively large benchmark repository of 150 open-source software applications, comprising approximately 13 million lines of code (as reported by CKJM Extended). From the final repository we made sure to exclude software projects that were also part of the repository that was used for the calibration of the model (see Section 3.2.1). In order to ensure diversity and avoid bias, the software applications were selected in a black-box manner, without filtering them based on their semantics (i.e., size, domain, creator, etc.). Hence, the resulting repository comprises highly diverse software applications with respect to their size, domain, and quality. This is important since the evaluation process should test the proposed model on different cases of software applications.

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

[18] https://find-sec-bugs.github.io/

Table 13 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), a security model that completely omits software metrics (i.e., SI'), and a quality model that is based primarily on software metrics (i.e., QI). Their corresponding rankings are also presented

Project Name	SI	SI Rank	SAVD	SAVD Rank	SI'	SI' Rank	QI	QI Rank
actframework	0.352024	133	2.363677	71	0.375633	125	0.194038	149
ansj-seg	0.457336	96	2.459829	74	0.431643	100	0.654175	46
apk-parser	0.583697	54	3.266809	95	0.557523	63	0.728607	25
AutoLoadCache	0.341156	136	6.531731	124	0.301215	136	0.615691	61
awaitility	0.454287	98	2.30052	68	0.432813	98	0.601281	64
azure-sdk-for-java	0.478949	91	6.849798	127	0.529085	68	0.173422	150
Cerberus	0.196344	148	16.933207	146	0.119288	148	0.659991	44
citrus	0.470902	93	3.290582	96	0.462166	93	0.508477	97
clarity	0.565259	57	3.892162	104	0.569655	58	0.537626	89
cloudhopper-smpp	0.229077	147	5.207646	118	0.202855	146	0.40112	126
commonmark-java	0.600761	48	1.976895	53	0.590902	52	0.653937	47
datastructure	0.342878	135	5.310971	119	0.270288	138	0.80466	7
dcm4che	0.442593	106	2.587713	78	0.482294	83	0.207073	148
disconf-demos-java	0.448576	101	14.954486	144	0.410673	110	0.720019	28
dockerfile-maven	0.508837	73	2.866502	85	0.476281	86	0.709367	31
docker-java	0.431021	113	6.938421	128	0.420548	107	0.463996	112
docker-maven-plugin	0.343779	134	2.133495	62	0.334386	133	0.441285	118
docx4j	0.635219	40	3.076223	90	0.682077	36	0.440397	120
easy-rules	0.402111	123	4.055343	106	0.35223	129	0.710395	30
ECharts	0.545777	60	3.415477	100	0.578902	57	0.321234	141
effective-java-examples	0.500633	77	3.989102	105	0.464882	90	0.758016	18
embedded-redis	0.384162	127	2.555366	77	0.339201	131	0.687792	38
EMV-NFC-Paycard	0.897857	12	1.010729	21	0.977198	11	0.438446	121
essentials	0.509864	72	2.356457	70	0.479288	84	0.699716	33
ews-java-api	0.493813	82	2.04748	56	0.529508	67	0.298887	144

Table 13 (continued)

Project Name	SI	SI Rank	SAVD	SAVD Rank	SI’	SI’ Rank	QI	QI Rank
Examination-System	0.442733	105	5.928509	122	0.429612	102	0.437661	122
fastdfs-client-java	0.485265	88	3.309291	98	0.484723	81	0.4905	104
fast-serialization	0.776316	26	1.645736	43	0.826841	26	0.493368	103
fastweixin	0.48152	90	4.183605	109	0.476895	85	0.547673	86
flexmark-java	0.924524	4	1.489728	35	0.994495	5	0.560878	75
flip-tables	0.371667	129	4.132231	107	0.283944	137	0.846516	3
FLogger	0.312713	140	8.154188	132	0.243971	141	0.746262	22
fluent-validator	0.264751	142	8.567662	134	0.214544	144	0.580868	68
flyingsaucer	0.888072	13	0.576908	11	0.986969	8	0.343078	138
fnlp	0.441396	108	4.466051	114	0.409936	111	0.667239	41
gecco	0.257109	144	7.309941	130	0.202556	147	0.587892	66
geofire-java	0.918956	5	1.453045	33	0.982995	9	0.533132	91
geohash-java	0.714379	34	0.8663	18	0.71695	33	0.658575	45
google-oauth-java-client	0.533626	68	2.931691	87	0.510792	70	0.684436	39
hbc	0.410178	120	4.194535	110	0.384466	120	0.565371	73
Hive2Hive	0.483311	89	10.206814	138	0.462735	91	0.63443	51
HotswapAgent	0.390755	124	1.533333	39	0.381015	123	0.440991	119
incubator-dubbo	0.341017	137	1.766615	47	0.317506	135	0.484794	107
infinitest	0.902785	6	0.856664	17	0.998669	2	0.399731	127
itchat4j	0.415945	118	8.811681	135	0.408676	113	0.521075	93
itextpdf	0.592183	52	1.010885	22	0.649083	41	0.262899	146
J2V8	0.632275	42	2.066902	59	0.671959	38	0.385134	131
jade4j	0.902218	8	0.103928	4	0.994247	6	0.406802	125
jansi	0.598783	49	1.061445	24	0.62205	43	0.477928	108
java-client-api	0.523719	70	1.352874	31	0.524074	69	0.549435	83
java-dns-cache	0.526045	69	3.818251	102	0.487677	80	0.785271	14

Table 13 (continued)

Project Name	SI	SI Rank	SAVD	SAVD Rank	SI’	SI’ Rank	QI	QI Rank
javaewah	0.699787	35	0.084814	2	0.736077	32	0.469625	110
java-faker	0.569881	56	1.20279	28	0.594224	51	0.452161	115
java-memcached-client	0.595624	50	2.066252	58	0.612473	46	0.505635	99
java-speech-api	0.485521	87	1.528013	37	0.475346	87	0.596666	65
jBCrypt	0.967839	1	0	1	0.977318	10	0.900093	2
jcabi-aspects	0.845957	17	0.798722	15	0.860987	20	0.800533	8
jdeb	0.541412	64	3.133351	92	0.53169	66	0.629569	52
jdonframework	0.102865	150	3.39736	99	0.026705	150	0.55602	78
jesque	0.556685	58	1.728866	46	0.566764	59	0.552479	81
jgit-cookbook	0.860868	15	1.498011	36	0.881539	18	0.792696	13
jHiccup	0.493888	81	4.225352	111	0.451041	95	0.796787	11
jieba-analysis	0.58456	53	2.751977	80	0.563133	60	0.7544	19
jitwatch	0.544635	61	2.836547	83	0.561145	61	0.45101	117
jmustache	0.263917	143	2.060439	57	0.204908	145	0.619848	58
jpinyin	0.72193	33	2.19106	63	0.7143	35	0.805136	6
json-simple	0.777122	25	0.485358	9	0.771248	29	0.795975	12
JSqlParser	0.771822	28	0.249231	7	0.84818	24	0.373559	133
Jupiter	0.434233	112	44.267632	149	0.416996	108	0.576045	70
jvmtop	0.273965	141	4.157339	108	0.246761	140	0.459641	113
jzmq	0.358069	132	2.276539	66	0.351583	130	0.392538	130
kilim	0.79681	23	0.90467	20	0.851078	22	0.506018	98
kryonet	0.856678	16	0.64683	13	0.912274	15	0.542199	88
lanproxy	0.491656	86	8.538633	133	0.454822	94	0.741176	23
lanterna	0.495402	79	1.59815	40	0.493085	77	0.536264	90
lavagna	0.506193	75	6.435867	123	0.501638	74	0.548856	85
librec	0.880626	14	1.079325	26	0.944146	14	0.510615	96

Table 13 (continued)

Project Name	SI	SI Rank	SAVD	SAVD Rank	SI’	SI’ Rank	QI	QI Rank
light-admin	0.534388	67	3.291401	97	0.506519	73	0.688091	37
light-task-scheduler	0.45567	97	2.958475	88	0.434122	97	0.615785	60
LinuxJavaFixes	0.82487	21	1.060178	23	0.88476	17	0.486358	106
Luyten	0.505584	76	1.997624	54	0.508273	72	0.51186	95
mango	0.419876	116	2.792263	81	0.395309	117	0.570565	71
marytts	0.553067	59	3.772219	101	0.585807	54	0.364127	135
metadata-extractor	0.730007	31	2.363935	72	0.747254	31	0.648217	48
Minim	0.817954	22	1.106283	27	0.855763	21	0.64148	49
mockserver	0.477356	92	2.395899	73	0.472208	88	0.560638	76
mp3agic	0.644456	38	1.633259	42	0.662067	39	0.471498	109
mybatis-3	0.253467	145	1.813956	48	0.222361	142	0.418656	124
mysql-binlog-connector	0.60821	46	2.261015	65	0.629682	42	0.486857	105
nanohttpd	0.424165	115	3.051317	89	0.414694	109	0.524222	92
natty	0.902403	7	0.097171	3	0.998311	3	0.398949	129
nlp-lang	0.593405	51	2.492794	75	0.601654	50	0.542723	87
obd-java-api	0.662797	36	1.368791	32	0.674689	37	0.629032	54
OpenID-Connector	0.442037	107	6.795628	126	0.447669	96	0.469031	111
openmessaging-java	0.534546	66	2.657807	79	0.494144	75	0.799871	10
open-replicator	0.784442	24	1.528384	38	0.824081	27	0.580997	67
opentracing-java	0.575996	55	2.866356	84	0.557547	62	0.712834	29
openwebflow	0.935827	3	0.542672	10	0.999833	1	0.604123	63
opsu	0.126092	149	2.278014	67	0.072071	149	0.373825	132
ormlite-core	0.362458	131	1.941268	52	0.367949	126	0.329487	140
oshi	0.622193	44	3.215105	94	0.609444	49	0.720641	27
paoding-rose	0.454234	99	8.962532	136	0.43269	99	0.626527	56
parallec	0.329874	138	10.328638	139	0.321387	134	0.433726	123

Table 13 (continued)

Project Name	SI	SI Rank	SAVD	SAVD Rank	SI’	SI’ Rank	QI	QI Rank
pcap4j	0.541731	63	1.244906	29	0.583662	55	0.302412	142
pcollections	0.843381	18	2.08717	61	0.867907	19	0.728448	26
pf4j	0.402288	122	13.391826	143	0.387875	119	0.501004	101
pollexor	0.60369	47	2.079002	60	0.611502	48	0.515531	94
prettytime	0.946105	2	0.640113	12	0.944575	13	0.954439	1
psi-probe	0.42465	114	12.625706	142	0.381166	122	0.680585	40
pushy	0.49364	83	157.889437	150	0.484165	82	0.618317	59
qart4j	0.767537	29	3.12256	91	0.769518	30	0.747407	20
QRGen	0.643101	39	0.199044	6	0.61211	47	0.799919	9
re2j	0.90167	10	0.119374	5	0.949749	12	0.552651	80
red5-server	0.386592	126	18.724715	147	0.36188	127	0.553256	79
RedisClient	0.901686	9	0.468268	8	0.997421	4	0.399484	128
restcountries	0.409211	121	24.830699	148	0.354596	128	0.738972	24
restx	0.52012	71	2.913895	86	0.510085	71	0.612934	62
Resty	0.449362	100	4.437426	113	0.430926	101	0.562211	74
rome	0.773322	27	2.310704	69	0.849344	23	0.351602	137
Saturn	0.389488	125	7.472227	131	0.396584	116	0.358374	136
sentry-java	0.412249	119	4.632893	116	0.383335	121	0.627771	55
signpost	0.321402	139	2.228976	64	0.25906	139	0.689506	36
simplenlg	0.500524	78	1.477832	34	0.493679	76	0.576995	69
smart	0.46726	94	11.790393	141	0.427343	103	0.697855	34
spring-roo	0.53518	65	1.667379	45	0.580025	56	0.302319	143
sqlite-jdbc	0.493073	85	2.813307	82	0.531702	65	0.259121	147
stateless4j	0.494501	80	1.828153	49	0.488287	79	0.548956	84
swagger-core	0.493445	84	2.026295	55	0.492698	78	0.504965	100
tabula-java	0.631983	43	1.864677	51	0.619789	44	0.701864	32

Table 13 (continued)

Project Name	SI	SI Rank	SAVD	SAVD Rank	SI'	SI' Rank	QI	QI Rank
takes	0.829594	20	0.831699	16	0.8341	25	0.836753	4
tess4j	0.374361	128	6.786662	125	0.338592	132	0.550173	82
thymeleaf	0.737764	30	1.075142	25	0.790333	28	0.45641	114
traccar	0.63278	41	0.872846	19	0.615746	45	0.746928	21
transmittable-thread	0.506995	74	4.662477	117	0.470369	89	0.760415	17
ttorrent	0.436991	111	9.18105	137	0.423953	104	0.565476	72
unirest-java	0.466348	95	1.658833	44	0.423361	105	0.696228	35
vlcj	0.368498	130	7.18059	129	0.391746	118	0.263526	145
vraptor4	0.419702	117	5.401097	120	0.378306	124	0.666213	42
webcam-capture	0.253044	146	5.551322	121	0.217595	143	0.495598	102
webdrivermanager	0.440957	109	16.768977	145	0.46272	92	0.336948	139
webmagic	0.447593	103	4.351055	112	0.421812	106	0.629542	53
weixin4j	0.54181	62	4.575237	115	0.54323	64	0.55829	77
weixin-popular	0.443848	104	3.187919	93	0.409644	112	0.620194	57
weixin-sdk	0.437277	110	11.148431	140	0.404992	114	0.638327	50
Word2VEC-java	0.615732	45	1.852995	50	0.586055	53	0.808148	5
XChart	0.651997	37	2.503507	76	0.652501	40	0.66576	43
xmemcached	0.831587	19	1.609042	41	0.907732	16	0.364774	134
ysoserial	0.44773	102	3.818674	103	0.398239	115	0.768416	16
ZhiHuSpider	0.900138	11	1.334701	30	0.987149	7	0.451387	116
zxing	0.722341	32	0.716788	14	0.715144	34	0.779348	15

applications, along with their *Security Indexes*, their *SAVDs*, and their corresponding rankings are presented in Table 13.

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) (Spearman, 1987), which is a non-parametric and non-sensitive to outliers statistical test. To interpret the strength of the observed correlation, the thresholds proposed by (Cohen, 2013) were used. According to (Cohen, 2013), 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 ($a = 0.05$).

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, 2013)) 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 acceptation 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 be reliably 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, as opposed to *SAVD* that is a "coarse-grained" indicator of internal software security, our proposed model provides finer-grained assessments. This holds because it groups the security-relevant static analysis alerts into vulnerability categories and calculates an individual *SAVD* for each one of these categories. Moreover, it considers security information retrieved from software metrics. Our proposed SAM uses more security-relevant information for determining the final security score of software products than the *SAVD* metric, hence the assessment results of the two approaches are not expected to perfectly match.

5.2 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 several iterations of aggregation that are performed for the calculation of the final security score. Hence, in order for the model to be reliable, the introduction (or removal) of even a trivial number of security-relevant issues between two versions of a software product should be reflected as a drop (or increase) in the produced *Security Index*. Hence, 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 source code changes with security implications and highlight these changes with a relevant change in the overall *Security Index*.

A potential approach for investigating the ability of the proposed model to reflect security changes is by applying it to the vulnerable and clean versions of a set of software applications. If the model tends to assign higher scores to the clean versions (or conversely lower scores to the vulnerable ones), it can be considered effective in reflecting security changes.

To this end, as an initial attempt, we decided to use the code examples that were used in a previous study (Siavvas et al., 2019a). In brief, in this study, four single-class subject applications with known vulnerabilities (i.e., Cross-site Scripting and OS Command Injection) were selected from the repositories of OWASP Benchmark[19]and NIST's Juliet Test Suite (Boland & Black, 2012), and their security issues were identified and manually fixed, by implementing the appropriate security mechanisms through code transformations. Subsequently, these applications were used in this study as the basis for measuring the energy footprint of the applied security mechanisms. It should be noted that the security issues were identified through manual code review and fixed based on guidelines provided by OWASP[20] and CWE[21]. Neither SAM nor PMD were used to guide these refactoring activities. These applications are available on the website with the supporting material of the paper (Online, 2020).

Since both the vulnerable and the clean versions of these applications are available, they render suitable candidates for our analysis. Hence, the proposed SAM was employed to calculate the *Security Indexes* of the vulnerable and clean versions of the selected applications. The results of this analysis are presented in Table 14.

As can be seen in Table 14, the code modifications that led to the removal of security issues lead to a significant increase in the *Security Index* of each application. More specifically, the *Security Index* of the vulnerable versions is 0.39 on average, whereas after the vulnerability elimination this score increases to 0.91 on average. In fact, a significant increase, more than 100% is observed in all four cases. This is reasonable, if we consider that the code modifications led to the removal of actual (not potential) vulnerabilities that reside in the code. Hence, the proposed SAM tends to assign higher security scores to the clean versions of the selected software applications. This provides evidence for the ability of the proposed SAM to reflect security-relevant code-level changes.

However, in order to reach safer conclusions, a large-scale analysis is required. This demands the construction of a large dataset containing both vulnerable and clean versions of a considerable number of software applications. Nevertheless, manually removing or injecting vulnerabilities (as we did in the aforementioned experiment) is a very

19 https://owasp.org/www-project-benchmark/

20 https://www.owasp.org/

21 https://cwe.mitre.org/

Table 14 The *Security Indexes* of the vulnerable and clean versions of four single-class subject applications retrieved from OWASP Benchmark and the Juliet Test Suite (Boland & Black, 2012). The removal of an actual vulnerability leads to a significant increase in the *Security Index* of a subject application

Subject	$SI_{vulnerable}$	SI_{clean}
CWE78_Juliet	0.39729	0.90309
CWE78_OWASP	0.44208	0.90251
CWE80_Juliet	0.36094	0.94608
CWE80_OWASP	0.36076	0.89956

time-consuming and effort-demanding process, whereas doing this for a very large number of applications is considered unrealistic. Hence, automating the process of constructing such a dataset is highly necessary for the purposes of the present experiment.

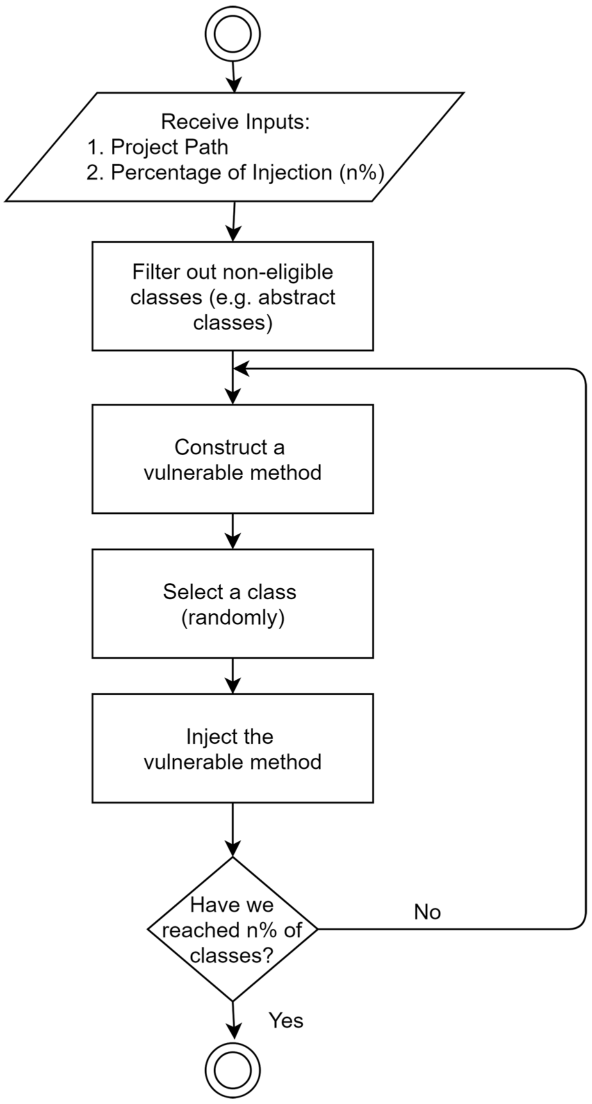
For the purpose, in order to automate the process of the dataset construction, and, in turn, conduct a more exhaustive experiment, we developed a dedicated vulnerability injector, i.e., a tool able to inject security-relevant issues (i.e., potential vulnerabilities) to the source code of a given software 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 15 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 Fig. 9 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.

Table 15 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

Fig. 9 The flow chart of the *Vulnerability Injector*


Receive Inputs:
1. Project Path
2. Percentage of Injection (n%)
Filter out non-eligible classes (e.g. abstract classes)
Construct a vulnerable method
Select a class (randomly)
Inject the vulnerable method
Have we reached n% of classes?
No
Yes

It then constructs a vulnerable method by randomly selecting a number of vulnerability issues from Table 15. An example of such a method is presented in Listing. 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).

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 three 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

Table 16 The *Security Indexes* of open-source applications after going through vulnerability injection

Project Name	$SI_{original}$	$SI_{1\%}$	$SI_{10\%}$	$SI_{25\%}$
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
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

Table 16 (continued)

Project Name	$SI_{original}$	$SI_{1\%}$	$SI_{10\%}$	$SI_{25\%}$
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
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
parallec	0.329874	0.319669	0.266816	0.242659

Table 16 (continued)

Project Name	$SI_{original}$	$SI_{1\%}$	$SI_{10\%}$	$SI_{25\%}$
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	0.71179	0.584382	0.468349
traccar	0.63278	0.599191	0.484344	0.387687
transmittable-thread-local	0.506995	0.468214	0.424198	0.38053
ttorrent	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
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

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 successfully compile 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 16.

As can be seen in Table 16, 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 is can be observed more clearly in the box plots presented in Fig. 10. 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.

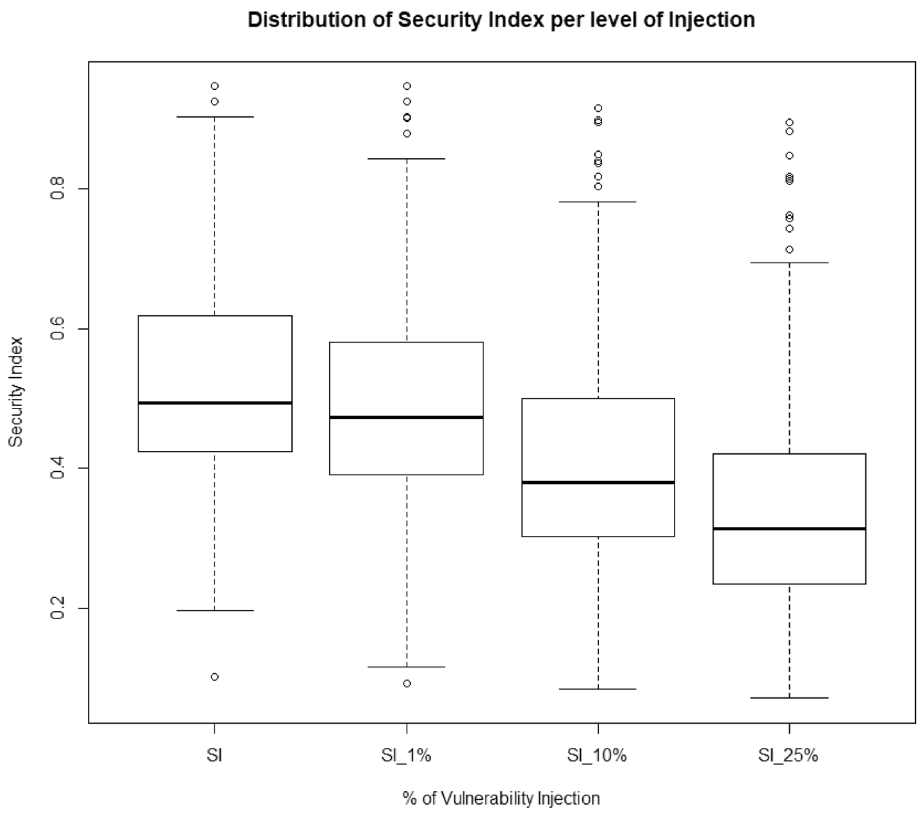


Fig. 10 The boxplots of the *Security Indexes* of each injection groups

Formal mathematical analysis is required, in order to reach safer conclusions. For this purpose, we decided to use 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 any statistically significant difference exists at least between the security scores of the original applications and 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 two or more samples 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 assumptions that should be satisfied in order to apply rANOVA are the following:

– ***Normality Assumption***: This assumption requires the pair-wise differences between the studied groups to follow a normal distribution.
– ***Sphericity Assumption***: This assumption requires the pair-wise differences between the studied groups to have equal variances.

In order to test the *Normality Assumption* we applied the *Shapiro-Wilk test*, while for validating the *Sphericity Assumption* we applied the *Mauchly's test*. The results of these experiments led us to the conclusion that these two assumptions are not fully satisfied for the data of the present experiment. The detailed results of these tests are provided on the web page with the supporting material of the present work (Online, 2020).

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.

Table 17 The results (i.e., *p-values*) of the post-hoc Nemenyi Test

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

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 17. As can be seen in Table 17, 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.

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 number[22] 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.

5.3 Comparison with a CVE-based approach

As mentioned in Section 5.1, in order to evaluate the capacity of the proposed SAM to reliably assess the security level of software projects, its results should be compared to the assessment results provided either by similar security models or by security experts. However, due to the lack of similar operationalized models and expert-based security assessments, in Section 5.1, we compared the proposed SAM with another popular security metric, i.e., the *SAVD* calculated using FindBugs (Heitlager et al., 2007).

Since both SAM and *SAVD* are CWE-based approaches, i.e., they are based on CWE issues detected by different static code analyzers, and in order to enhance the completeness of our evaluation, in the present section, we compare the proposed SAM with a different non-CWE-based approach. More specifically, we assess the security level of a set of software projects based on relevant vulnerability information that can be found in the *Common Vulnerability and Exposures (CVE)*[23] database, and we compare these assessments with the assessments provided by SAM. A close correlation between these assessments will provide further confidence for the ability of the proposed SAM to reflect the security of software projects.

For the purposes of the present experiment, the *OWASP Dependency Check*[24] tool was used as the reference point of our analysis. The *OWASP Dependency Check* is a software composition analysis tool that attempts to detect publicly disclosed vulnerabilities contained within the dependencies of software projects. More specifically, the tool receives a software project as input and checks whether it has links to known vulnerabilities that have been disclosed in the *National Vulnerability Database (NVD)*[25], and that are uniquely identified by relevant publicly available CVE entries (i.e., identifiers). After the analysis is complete, the tool produces detailed reports with the assessment results, which contain the

[22] It should be noted that even the 25% injection of security issues is considered small. 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 insecure instructions, which is trivial compared to the overall size (i.e., lines of code) of the application.

[23] CVE is a list of publicly disclosed cybersecurity vulnerabilities that is free to search, use, and incorporate into products and services (link: https://cve.mitre.org/cve/).

[24] https://owasp.org/www-project-dependency-check/

[25] https://nvd.nist.gov/

number of the identified vulnerable dependencies, along with the exact CVEs (i.e., CVE identifiers) of the detected vulnerabilities.

For the present analysis, the same benchmark repository of real-world software applications that was used in the previous analyses (i.e., in Section 5.1 and in Section 5.2), comprising 150 popular open-source Java applications, was utilized. The *OWASP Dependency Check* tool was manually applied to each one of the software projects of this benchmark. Only those applications that were able to be analyzed with the *OWASP Dependency Check* tool were selected for the final analysis. In fact, from the original 150 software projects, the *OWASP Dependency Check* tool was able to generate reports for 111 of them. For the rest of them, the tool was not able to parse their dependencies or their list of dependencies was missing, making it unable to search on the NVD for relevant CVE entries for these applications. The detailed HTML reports of the aforementioned analysis are available on the web page with the supporting material of the present paper (Online, 2020). These reports contain detailed information about the CVEs that were detected for real-world projects, and therefore we believe that this information may be useful for further research endeavors.

The HTML reports that were produced by the *OWASP Dependency Check* tool were then parsed, and the total number of the identified CVEs was computed (i.e., *#CVEs*) for each one of the analyzed software applications (i.e., projects). In this experiment, we used the number of the identified CVEs of a software project as an indicator of its security level, as they correspond to security issues that the projects contain. The *Security Indexes* of these projects were already available from the analysis that was performed in Section 5.1. Next, we obtained two individual rankings of the selected software projects, one based on their *Security Indexes*, and another one based on their number of identified CVEs (i.e., *#CVEs*) that were determined by the *OWASP Dependency Check* tool. A subset of the analyzed software applications, along with their *Security Indexes*, their *#CVEs*, and their corresponding rankings are presented in Table 18. The full table with the results is available on the web page with the supporting material of the present work (Online, 2020).

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. Similarly to Section 5.1, we decided to use the *Spearman's rank correlation coefficient* (ρ) (Spearman, 1987), which is a non-parametric and non-sensitive to outliers statistical test. To interpret the strength of the observed correlation, the thresholds proposed by (Cohen, 2013) were used. 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), which was tested at the 95% confidence level ($a = 0.05$):

H_0: *No statistically significant correlation is observed between the two rankings.*
H_1: *A statistically significant correlation is observed between the two rankings.*

The calculated *Spearman's rank correlation coefficient* between the two rankings was found to be $\rho = 0.63$, which is a positive and strong (according to (Cohen, 2013)) 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, indicating 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 number of their CVEs that were detected by the *OWASP*

Table 18 A fragment of the security evaluation results of the selected open-source software products, as produced by the proposed security assessment model (i.e., SI), the *OWASP Dependency Check*tool (i.e., number of CVEs), along with their corresponding rankings. The complete table is available online (Online, 2020)

Project Name	Security Index	SI Rank	#CVEs	CVE Rank
jBCrypt	0.967839	1	0	1
fastweixin	0.48152	64	7	71
Resty	0.449362	73	60	93
citrus	0.470902	67	27	83
AutoLoadCache	0.341156	101	63	95
rome	0.773322	17	6	68
weixin4j	0.54181	46	34	86
librec	0.880626	6	4	57
ysoserial	0.44773	74	30	85
signpost	0.321402	104	27	83
smart	0.46726	68	141	109
azure-sdk-for-java	0.478949	65	118	106
Jupiter	0.434233	83	21	82
red5-server	0.386592	96	80	99
mockserver	0.477356	66	102	102
incubator-dubbo	0.341017	102	142	110
mango	0.419876	87	13	77
EMV-NFC-Paycard	0.897857	5	1	35
OpenID-Connector	0.442037	79	103	103
dockerfile-maven	0.508837	52	59	92
dcm4che	0.442593	78	99	101
qart4j	0.767537	19	2	45
psi-probe	0.42465	85	121	107
light-task-scheduler	0.45567	70	132	108
tess4j	0.374361	98	9	73
swagger-core	0.493445	62	50	90
itchat4j	0.415945	89	5	65
parallec	0.329874	103	67	96
lavagna	0.506193	54	115	104
sentry-java	0.412249	90	19	79
docker-maven-plugin	0.343779	100	10	76
google-oauth-java-client	0.533626	50	19	79
vraptor4	0.419702	88	9	73
apk-parser	0.583697	39	3	55
Saturn	0.389488	95	362	111
re2j	0.90167	4	0	1
java-memcached-client	0.595624	36	16	78
Cerberus	0.196344	109	116	105
weixin-sdk	0.437277	81	97	100
Examination_System	0.442733	77	71	98
jdeb	0.541412	48	8	72
restcountries	0.409211	92	19	79

Table 18 (continued)

Project Name	Security Index	SI Rank	#CVEs	CVE Rank
docker-java	0.431021	84	68	97
webcam-capture	0.253044	107	37	87
jesque	0.556685	43	49	89
natty	0.902403	2	0	1
fluent-validator	0.264751	105	9	73
java-client-api	0.523719	51	43	88
paoding-rose	0.454234	72	55	91
webmagic	0.447593	75	60	93

Dependency Check tool (*#CVEs*). 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, which, contrary to the analysis of Section 5.1 is not CWE-based, providing support for its ability to reliably reflect the security level of software products. Hence, the results of the present analysis along with the results of Section 5.1, provide confidence for the capacity of the proposed SAM to reflect the security level of software products in practice.

Similarly to Section 5.1, it should be noted that the two rankings were not expected to be identical, since the two approaches are based on different security-related information. More specifically, the proposed SAM is based on the CWE issues that are identified through static analysis, whereas the approach that was used as the reference point of the present experiment was based on the CVE entries that are detected by the *OWASP Dependency Check* tool. Hence, the assessment results that are produced by the two approaches are not expected to perfectly much. However, the existence of a statistically significant strong correlation between these assessments provides confidence that the proposed SAM is able to reflect the security level of software products, or at least that it does not provide information that is irrelevant to the security of the analyzed software.

5.4 Class-level security assessment

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. Another interesting problem, 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 (i.e., predict) the existence of software vulnerabilities in this 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 class-level vulnerability prediction (i.e., for highlighting potentially vulnerable classes).

5.4.1 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, existing datasets are highly unbalanced, while they are constructed based mainly on reported vulnerabilities of software products (e.g., (Scandariato et al., 2014; Zhang et al., 2019)). 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.

OWASP Benchmark 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 (Boland & Black, 2012)), 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, whereas those containing false positives as the clean components of the present analysis.

The proposed SAM was employed in order to calculate the security scores 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 removing 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.

5.4.2 Correlation analysis

In order to investigate the ability of the proposed SAM to provide reliable class-level estimations of software security, we initially applied correlation analysis, with the purpose to examine the relationship between the produced *Security Index* and the existence of vulnerabilities in software classes. A statistically significant relationship would provide confidence for the ability of the proposed model to indicate the existence of vulnerabilities in software classes.

To this end, the proposed SAM was initially employed to compute the *Security Indexes* of the OWASP Benchmark classes that were selected through the process described in Section 5.4.1. In order to enhance the completeness of the present analysis and reach more reliable conclusions, apart from SAM, we also included the *SAVD* metric in our analysis. More specifically, we calculated the *SAVDs* of the selected subset of OWASP Benchmark classes, by employing the FindBugs static code analyzer, using the same configuration that was used for the experiment that was presented in Section 5.1. A small fragment of the analyzed classes, along with their *Security Indexes* and *SAVDs* that were computed based on the aforementioned procedure are presented in Table 19. The purpose of this fragment is to demonstrate the structure of the dataset used for the present analysis. The complete results are available on the website with the supporting material of the present work (Online, 2020). It should be noted that the last column of Table 19 is the *vulnerability state* of the class, which denotes whether the corresponding class is vulnerable (i.e., 1), or clean (i.e., 0).

Subsequently, we calculated the correlation between the *Security Index* and the *vulnerability state* (i.e., ground truth), as well as between the *SAVD* and the *vulnerability state*. For the purposes of the present study, we decided to use the *Point-biserial correlation coefficient* (r), which is used for computing the correlation between a continuous and a dichotomous (i.e., binary) variable. For the characterization of the correlation strength, similarly to the previous experiments, we used the thresholds suggested by (Cohen, 2013). However, it should be noted that we do not expect the correlations to be strong. Even a weak correlation, is normally acceptable for providing evidence for the potential ability of a factor to indicate the existence of vulnerabilities (e.g., (Shin & Williams, 2008b; Chowdhury & Zulkernine, 2011; Shin et al., 2011; Medeiros et al., 2017; Siavvas et al., 2017b; Moshtari et al., 2013; Moshtari & Sami, 2016; Stuckman et al., 2017; Ferenc et al., 2019; Jimenez et al., 2019; Zhang et al., 2019)).

The *Point-biserial correlation coefficient* (r) between the *Security Indexes* of the classes and their *vulnerability state* was found to be $r = -0.31$, which is a negative and medium

Table 19 The *Security Indexes* (i.e., SI), the *SAVDs*, and the *Vulnerability States* of a fragment of the constructed vulnerability dataset, which contains 1200 classes. The SI' column contains the *Security Indexes* of the selected classes that were computed by an alternative security assessment model that omits software metrics. The complete table is available online (Online, 2020)

Class Name	SI	SAVD	SI'	Vulnerability State
Bench_00126	0.794198	29.411764	0.764353	0
Bench_01564	0.732225	22.304832	0.695615	0
Bench_01429	0.444389	35.087719	0.395615	1
Bench_01408	0.358902	30.120481	0.287730	1
Bench_01012	0.637875	38.461538	0.585201	0
Bench_00866	0.389418	18.691588	0.323054	0
Bench_00551	0.470953	42.253521	0.391098	1
Bench_01039	0.568046	28.089887	0.517363	0
Bench_01041	0.542754	25.906735	0.488677	1
Bench_00405	0.689363	44.247787	0.640945	0
Bench_02026	0.713227	27.777777	0.668393	0
Bench_01648	0.671609	51.724137	0.620549	1
Bench_00535	0.495721	20.618556	0.435758	0
Bench_00947	0.510003	27.027027	0.438829	1
Bench_01479	0.433639	23.584905	0.366153	0
Bench_01740	0.695840	15.686274	0.655495	1
Bench_02567	0.416972	20.408163	0.349303	1
Bench_00788	0.399086	20.979020	0.332508	1
Bench_01825	0.774997	23.333333	0.741085	0
Bench_01442	0.361662	27.210884	0.286471	1

correlation (according to (Cohen, 2013)). This correlation is statistically significant, since the *p-value* was found to be 4.945×10^{-11}, which is significantly lower than the threshold of 0.05. It should be noted that the negative value denotes that lower the *Security Index* the higher the probability for the associated class to be vulnerable (i.e., 1), which is the desired behavior.

Similarly, the *Point-biserial correlation coefficient* (r) between the *SAVDs* of the classes and the *vulnerability state* was calculated and found to be $r = 0.1$, which is a positive and weak correlation (according to (Cohen, 2013)). The *p-value* was found to be 3.73×10^{-11}, which suggests that the observed correlation is statistically significant. The positive correlation denotes that the higher the *SAVD* of a given class, the higher its probability to be vulnerable (i.e., 1), which is the desired behavior.

A direct comparison between the calculated correlations (in terms of absolute values) reveals that the *Security Index* is more closely related to the existence of vulnerabilities compared to the *SAVD* computed based on the FindBugs static code analyzer. This suggests that the proposed SAM may provide more reliable class-level security estimations compared to its counterpart (at least for the given dataset), despite the fact that it is based on a popular tool that is known for its ability to detect security issues. In addition to this, the statistically significant medium correlation provides evidence for the potential ability of the proposed model to indicate the existence of vulnerabilities in software classes, and therefore to be used as the basis for vulnerability prediction. However, since the observed correlation was not found to be strong, discriminant analysis is required in order to reach safer conclusions, which is presented in Section 5.4.3.

To further enhance the completeness of the present analysis, similarly to Section 5.1, we also computed the correlation between the ranking of the selected OWASP Benchmark classes based on their *Security Index*, and their ranking based on their *SAVD*. Similarly to Section 5.1, we decided to use the *Spearman's rank correlation coefficient* (ρ) (Spearman, 1987), which is a non-parametric and non-sensitive to outliers test. The thresholds proposed by (Cohen, 2013) were also used for interpreting the strength of the observed correlation. In order to reach safer conclusions with respect to the statistical significance of the observed relationship, the following hypothesis was formulated and tested at the 95% confidence level ($\alpha = 0.05$):

H_0: *No statistically significant correlation is observed between the two rankings.*
H_1: *A statistically significant correlation is observed between the two rankings.*

The calculated *Spearman's rank correlation coefficient* (ρ) between the two rankings was found to be $\rho = 0.19$, which is a positive, but weak correlation. Since the *p-value* was found to be lower than the threshold value of 0.05 (in fact, $p - value < 2.2 \times 10^{-12}$), the null hypothesis is rejected. This suggests that the observed correlation between the two rankings is statistically significant. The results of this analysis are in line with the results of the *Point bi-serial correlation*. More specifically, the two rankings were found to be correlated in a statistically significant manner, which is reasonable as both the *Security Indexes* and the *SAVDs* of the selected classes demonstrated a statistically significant correlation with the *vulnerability state* of these classes (i.e., ground truth). In addition to this, the *Security Indexes* demonstrated a much stronger correlation with the ground truth compared to the *SAVDs*, which explains the weak correlations between their rankings. Hence, this analysis further supports the observation that the *Security Index* may be a better vulnerability indicator than the FindBugs-based *SAVD*, and therefore that it can be used as the basis for vulnerability prediction. However, as already mentioned, in order to reach safer conclusions, discriminant analysis is also required.

5.4.3 Discriminant analysis

In order to investigate the ability of the proposed SAM to discriminate between vulnerable and clean software components, we employed discriminant analysis. 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 (Fig. 11).

Figure 11 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 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., (Shin & Williams, 2008; Munaiah & Meenley, 2016; Jimenez et al., 2019)). In particular, the following null hypothesis (along with its corresponding alternative hypothesis) was formulated and tested with confidence level 95% (i.e.,$a = 0.05$):

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.*

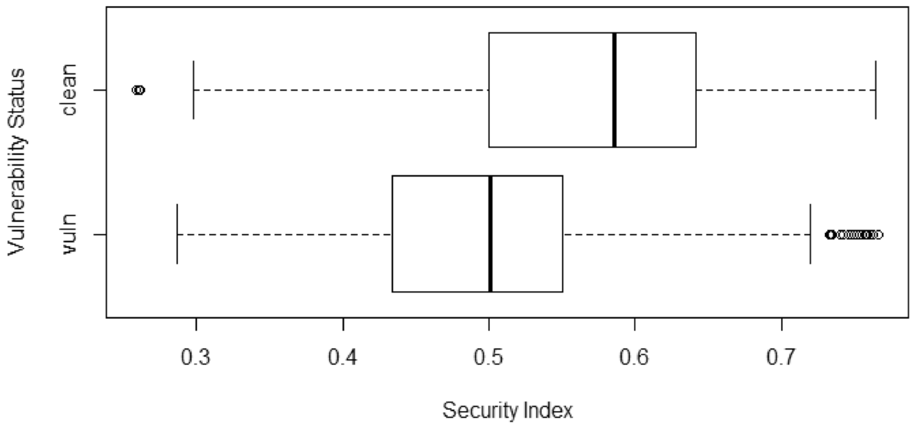


Fig. 11 Distribution of the *Security Indexes* of vulnerable and clean software components (i.e., classes)

After performing the test the *p-value* was found to be much lower than the threshold of 0.05 (i.e., *p-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.

For reasons of completeness, we also performed a *Wilcoxon Rank Sum* test between the *SAVDs* of the vulnerable and clean software components of the selected benchmark, in order to observe whether a statistically significant difference exists between their values. The test resulted in a *p-value* that was lower than the threshold value of 0.05 (*p-value* $< 2.2 \times 10^{-16}$), leading to the rejection of the null hypothesis. This suggests that the *SAVDs* that are calculated using FindBugs are also able to discriminate between vulnerable and clean software components, meaning that the FindBugs-based *SAVD* can potentially be used as a vulnerability indicator. However, since the *Security Index* demonstrated a much higher correlation to the *vulnerability state* of the selected classes compared to the FindBugs-based *SAVD*, it can be considered a stronger vulnerability indicator (at least for the studied dataset, i.e., the OWASP Benchmark).

In brief, since the proposed SAM 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 vulnerability prediction model, able to highlight classes of a software product that are potentially vulnerable. For instance, a specific threshold can be trained for the *Security Index* of the model, below which, a class will be considered vulnerable. However, it is obvious that a more elaborate study is required for the latter, which is left as a future work.

5.4.4 Vulnerability prediction based on the security index

In the previous sections (i.e., Section 5.4.2 and Section 5.4.3), correlation and discriminant analysis were performed, in order to check whether the *Security Index* produced by the proposed SAM can be used as a good indicator of software vulnerabilities. The results of these analyses showcased that the *Security Index* is closely correlated to the existence of vulnerabilities and that it has sufficient discriminative power. Hence, being in line with the literature, the preliminary results of these analyses provide us with confidence that the *Security Index* can be considered an indicator of software vulnerabilities, which can be used as the basis for constructing vulnerability prediction models.

In fact, the research in the field of vulnerability prediction focuses on (i) identifying potential indicators of software vulnerabilities (i.e., through empirical studies focusing on correlation and discriminant analysis) (Shin & Williams, 2008a, b; Chowdhury & Zulkernine, 2010; Siavvas et al., 2017b; Zhang et al., 2019; Jimenez et al., 2016; Sultana et al., 2017; Roumani et al., 2016; Medeiros et al., 2017; Sultana et al., 2019), and/or on (ii) building vulnerability prediction models (VPMs) based on the identified indicators, which are normally built based on machine learning (ML) techniques (Shin et al., 2011; Chowdhury & Zulkernine, 2011; Scandariato et al., 2014; Moshtari & Sami, 2016; Dam et al., 2018; Siavvas et al., 2020a; Kalouptsoglou et al., 2020).

Although the correlation and discriminant analyses are normally enough for detecting potential vulnerability indicators, in order to enhance the completeness of the present experiment, since the correlation and discriminant analysis showed that the *Security Index* can be considered a sufficient indicator of software vulnerabilities, we go one step further and we attempt to construct a uni-variate VPM based on this indicator. The construction of a VPM with sufficient accuracy would provide further confidence for the ability of the produced SAM to provide reliable class-level security assessment, and, in turn, that the *Security Index* can be used as the basis of vulnerability prediction. However, it should be clearly stated that examing the feasibility of constructing VPMs using the *Security Index* of the proposed SAM is out of the scope of the present paper. This analysis is performed in order to gain further insight with respect to the capacity of the proposed SAM to discriminate between vulnerable and clean classes, and therefore to provide reliable class-level security assessments.

The first step of our analysis is the selection of the dataset for the construction of the VPMs. For the purposes of the present experiment, the highly balanced dataset of vulnerable and clean components that was constructed in Section 5.4.1 based on the OWASP Benchmark was utilized. Since the produced models are uni-variate, the calculated *Security Index* was used as the independent variable, whereas the *vulnerability state* of each class of the dataset was used as the dependent variable (see Table 19). Based on this dataset, several models were built, using both linear and non-linear machine learning algorithms. In particular, we used the following ML algorithms:

- **Logistic Regression** is a classifier that predicts the probability of a categorical target variable Y belonging to a certain class by employing a logit function. Although the logit function makes logistic regression suitable for binary classification where there are two classes, it can be extended to support classification where multiple classes are present.
- **K-Nearest Neighbours** is a simple algorithm that initially keeps all available labeled data points in the memory. Once a new data point comes in, it gets classified based on the majority label of the k data points closest to it. The closeness between data points is computed by using a distance function (e.g., Euclidean distance).
- **Naïve Bayes** is a probabilistic classifier that is based on the Bayes' theorem. To make classifications, it computes the odds of a data point to belong into a specific class. Although Naive Bayes is simple and intuitive, it works under the assumption that all features are independent and they not affect the other, which is rarely the case in real-life classification tasks.
- **Support Vector Machine** is a classifier that tries to find the optimal N-dimensional hyperplane (i.e., support vectors) that maximizes the margin between the data points, thus making them distinctly separable. To achieve this, it tries to learn a non-linear function by linearly mapping the data points into high-dimensional feature space.
- **Random Forest** is a classifier that is constructed based on multiple decision trees. For the classification, the new instance (i.e., input vector) is fed as input to each one of the decision trees of the Random Forest, which predicts its class. Then the Random Forest collects all the votes that are produced by its decision trees and provides a final classification. Usually, the class that was selected by the majority of the decision trees is chosen as the final class of the new instance.
- **XGBoost** is a decision-tree-based ensemble ML algorithm that uses multiple decision trees to predict an outcome based on a gradient boosting framework.

The predictive performance of these models was evaluated and compared using popular performance metrics, namely Recall, Precision, and F1 score. For the evaluation, the 10-fold cross-validation approach was employed. According to this approach, the original dataset is randomly split into 10 folds, nine of them are used for training and one for testing. This process is repeated until each one of the 10 folds is used as a test set. The performance metrics of each one of the 10 folds are recorded and then the average of these 10 recorded metrics is computed. The average values of these performance metrics through these 10 folds are selected as the performance metrics of the model. This process was repeated for each one of the selected ML algorithms, and their predictive performance was compared based on the calculated performance metrics. The results of the evaluation of the produced models are presented in Table 20.

As can be seen in Table 20, the Logistic Regression is the best performing model, as it showcases an F1 Score of 0.71. This suggests that a uni-variate logistic regression model can predict the existence of vulnerabilities in software components with sufficient accuracy. Apart from Logistic Regression, the XGBoost also demonstrates high Recall, and a satisfactory F1 Score. It should be also noted that all the studied models demonstrate a predictive performance, which is better than the base model of random guess.

Hence, from the results presented in Table 20, it is eminent that the *Security Index* can be used as the basis for the construction of uni-variate VPMs with sufficient predictive performance, at least for the given dataset. The preliminary results of this analysis provide more confidence for the capacity of the proposed SAM to provide reliable class-level security assessments, and to be utilized for class-level vulnerability prediction.

The utilization of additional vulnerability indicators along with the *Security Index* is expected to lead to multi-variate VPMs with better predictive performance. More specifically, the *Security Index* could be used in conjunction with other factors for building more accurate VPMs, whereas it could probably enhance the predictive performance of existing VPMs that are based on other known vulnerability indicators (e.g., text-mining features). However, as already mentioned, this is out of the scope of the present paper, and it is left as a direction for future work.

5.5 Impact of software metrics

As mentioned in Section 3.1, 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

Table 20 Cross-validation averaged scores for all the produced class-level vulnerability prediction models

Classifier	Precision	Recall	F1 Score
Random Forest	0.59	0.58	0.58
Logistic Regression	0.66	0.76	0.71
K-NN	0.59	0.65	0.62
Naïve Bayes	0.64	0.71	0.67
XGBoost	0.62	0.76	0.68
SVM	0.64	0.68	0.66

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%, as discussed in Section 3.2.2), in order to convince the reader about our choice to include software metrics, a formal evaluation of their actual impact on the overall security assessment is provided in the present section.

5.5.1 Comparison with a security model that omits software metrics

As a first step towards this formal evaluation, the assessment results of the proposed model need to be compared to 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 alert-based properties (i.e., vulnerability categories). The thresholds of the alert-based properties of the new model are the same with those of the original model (see Table 7), while its weights were elicited based on the same pair-wise comparison matrices (e.g., Fig. 6), 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 5.1), 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 13. 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) (Spearman, 1987) was employed, and the following null hypothesis (along with its corresponding alternative hypothesis) was formulated and tested at the 95% ($a = 0.05$) 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-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. This suggests that the security assessments provided by the two models are highly similar.

To enhance the completeness of the present study, an additional analysis based on the OWASP Benchmark was conducted, in order to also compare the models with respect to their class-level assessments. More specifically, in Section 5.4, a vulnerability dataset was constructed based on the OWASP Benchmark and analyzed using the proposed SAM. For the purposes of the present experiment, we analyzed the same vulnerability dataset using the alternative model that omits software metrics. The *Security Indexes* of a fragment of these classes that were produced using this alternative model are presented in the fourth column of Table 19. The detailed results are available on the website with the supporting material of the present experiment (Online, 2020).

Subsequently, the *Point-biserial correlation* (r) between the *Security Indexes* that were calculated using the alternative model and the vulnerability state of the analyzed classes was computed. The value of the correlation was found to be $r = -0.32$, which is a negative and medium correlation according to (Cohen, 2013). In addition, this correlation is statistically significant, since its *p-value* was found to be 2.6×10^{-12}, which is significantly smaller than the threshold of 0.05. This correlation is almost equal to the correlation that was observed between the *Security Indexes* produced by the original SAM and the vulnerability states of the studied classes that that was calculated in Section 5.4.2, which was found to be $r = -0.31$. Hence, this suggests that both models are almost equally correlated to the ground truth (as reflected by the OWASP Benchmark), which indicates that the two models provide similar class-level security assessments, providing further support that they can be used interchangeably for security assessment.

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 13 and Table 19), the overall assessment (i.e., ranking) is almost the same, whereas their correlation with a ground truth of software vulnerabilities (as expressed by the OWASP Benchmark) is also almost equal. 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.

5.5.2 Comparison with a quality model

Although the previous analysis highlighted that the impact of the selected software metrics on the proposed model is very small (which is what we wanted), another interesting question is to investigate what observations could be made if this impact was high. For this purpose, we created a new model by manually setting its weights so that 80% of the produced score to be determined by the selected object-oriented (OO) software metrics, and only 20% by the defined weakness categories. Since the final score is determined predominantly by OO metrics, we treat the resulting model as a quality model (and not as a security model), and we term the produced score as *Quality Index (QI)*, instead of *Security Index (SI)*, as OO metrics are commonly treated in the literature as quality indicators (e.g., (Bansiya & Davis, 2002; Heitlager et al., 2007; Wagner et al., 2015; Siavvas et al., 2017b)).

Subsequently, similarly to the previous analysis, we computed the *Quality Indexes* of the 150 GitHub applications that were used previously, and ranked them based on these quality scores. The *Quality Indexes* and the corresponding ranking of these applications is presented in Table 13. Then, similarly to the previous analysis, we calculated the *Spearman's Rank Correlation Coefficient* (ρ) between this ranking and the ranking that was obtained based on the *Security Indexes* that were computed using the original model, which was found to be $\rho = 0.0665$. This is a very weak correlation (according to (Cohen, 2013), which is close to zero. In addition, the *p-value* was found to be 0.41875, indicating that the observed correlation is not statistically significant. Hence, the results of this analysis suggest that the two approaches provide different assessments, which provides further support to our decision to assign small weights to the metric-based properties, as higher weights would affect the overall assessment.

To reach safer conclusions we also employed the quality model in order to calculate the *Quality Indexes* of the classes of the OWASP Benchmark, and computed their correlation with their vulnerability state using the *Point-biserial correlation coefficient* (r). The detailed results of this analysis are available on the website with the supporting material of the paper (Online, 2020). The correlation was found to be $r = -0.17$, and its *p-value*=0.0016. This is a statistically significant negative and weak correlation (according to (Cohen, 2013)). The negative correlation indicates that the *Quality Index* tends to be lower for vulnerable classes, which is the desired behavior. Hence, the OO metrics-based quality model is not sufficiently correlated to the ground truth (as expressed by the OWASP Benchmark), as opposed to the other two security models (i.e., the original and the one that omits software metrics), which were found to have a medium correlation with the ground truth. This suggests (at least for the studied dataset) that software metrics are only weakly correlated to vulnerabilities. This is in line with the large body of knowledge, which suggests that software metrics are only weak indicators of vulnerabilities (Shin & Williams, 2008b; Chowdhury & Zulkernine, 2011; Shin et al., 2011; Medeiros et al., 2017; Siavvas et al., 2017b; Moshtari et al., 2013; Moshtari & Sami, 2016; Stuckman et al., 2017; Ferenc et al., 2019; Jimenez et al., 2019; Zhang et al., 2019).

5.5.3 Discussion

To conclude, the results of the present analysis justifies our decision to assign small weights to software metrics, since they were found to be only weakly correlated to vulnerabilities, whereas assigning higher weights was found to significantly affect the assessment results. In addition, the proposed security model was found to provide similar assessment results with an equivalent security model that omits software metrics, which provides confidence for its objectivity, and indicates that these models can be used interchangeably for security assessment. Although we recommend the adoption of the security model with the software metrics since it is more complete, we also provide the model that is based only on alert-based properties on the web page with the supporting material of the present paper (Online, 2020).

6 Static analysis, model transparency, and root-cause analysis

As mentioned in the previous sections, the proposed SAM is based exclusively on static analysis. This allows the proposed SAM to be applied regularly during the software development process from the very early stages of the implementation in a fully automated way, as static analysis does not require the execution of the software product under assessment. Despite the obvious benefits that SAM has from the adoption of static analysis, static analysis itself could benefit from the concepts and mechanisms that are provided by SAM.

One of the main shortcomings of static code analyzers, which hinders their adoption in practice, is their poorly presented results (Johnson et al., 2013; Siavvas et al., 2018a), which comprise long lists of raw warnings (i.e., alerts) or absolute values of software metrics. Firstly, although these results contain valuable security-related information about the

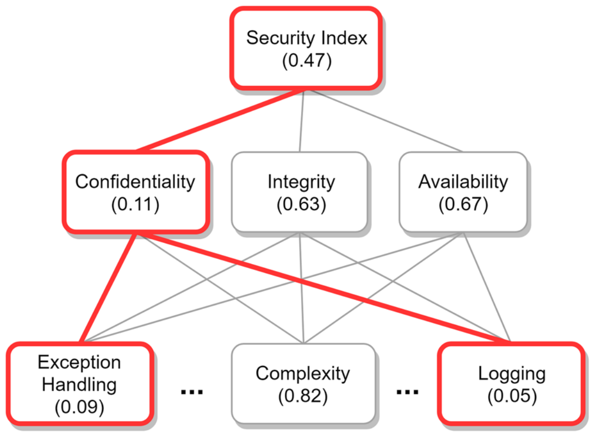


Fig. 12 An example demonstrating how the proposed model can be used for conducting root-cause analysis. A developer can start from the high-level security score of a given product (or class), and drill down to the code, in order to identify what issues led to the assignment of this score. In the given example, the path (i.e., sub-graph) that attracts the attention of the developer is marked with red color

product at hand, this information is difficult (if not impossible) to comprehend, especially by stakeholders with little or no technical knowledge, such as project managers. Another issue is that these long lists of warnings that are produced by the tools are difficult to inspect (a time-consuming & effort-demanding process that is often termed triaging (Walden & Doyle, 2012; Ruthruff et al., 2008)), which discourages developers from using them in practice.

The proposed SAM attempts to address the former issue (i.e., difficult-to-comprehend results), by aggregating the raw static analysis results into higher-level security scores, through several levels of aggregation. These security scores are more intuitive and easily understandable even by stakeholders with little or no technical knowledge. In addition to this, the hierarchical structure of the security model allows the stakeholders to focus on a specific level of abstraction, putting their focus on what matters most. For instance, a project manager could check only the overall security score (i.e., the *Security Index*) of the software under development, in order to get an overview of its overall security status, or they could check the security score of a specific security characteristic (e.g., *Confidentiality*, *Integrity*, and *Availability*) in order to see whether there are static analysis issues that affect this security characteristic.

More interestingly, the proposed SAM also contributes towards addressing the latter issue (i.e., impracticality of static analysis), through its hierarchical structure and transparency. In fact, the transparency of the proposed model enables the conduction of *root-cause analysis*, meaning that it can be used for identifying the underlying reasons that led to the assignment of a specific score to a given software product (or class). To better explain this concept, a simple example is illustrated in Fig. 12. Suppose that a specific product (or class) received a security score of 0.47, which is relatively low. As can be seen in Fig. 12, we can follow the model's hierarchy in order to gain more detailed insight. More specifically, by focusing on the layer of characteristics, we can see that the product received a low score in the characteristic of *Confidentiality*, which indicates that it may not handle well sensitive information. If we go one step further and examine the model's properties, we can see that very low scores were assigned to the properties of *Logging* and *Exception Handling*. This suggests that the given product (or class) suffers from many logging and exception handling issues, which may have an impact on its *Confidentiality*, and, in turn, its overall security.

From the above analysis, it is clear that the developer can start from the high-level security score of a given product (or class), and drill down to the code, in order to identify what issues led to the assignment of this score. Hence, the developers can actually use this information to better prioritize their testing and fortification efforts, for example by starting their refactoring activities by fixing these issues first. This suggests that apart from a high-level score, the model also facilitates the identification of low-level security problems that a program may have, which can be used to drive refactoring activities. Therefore, instead of triaging the long list of the produced static analysis warnings and fixing those issues in an arbitrary manner, developers can focus on subsets of this list that are more critical from a security viewpoint, and start their refactoring activities from these issues first.

7 Conclusion

In the present paper, we proposed a fully automated hierarchical Security Assessment Model (SAM) that is able to quantify the internal security level of software applications 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 (ISO, 2011) 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, as well as on knowledge retrieved from the Common Weakness Enumeration (CWE). 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 (Saaty, 2008) and SMARTS/SMARTERS (Edwards & Barron, 1994) 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 level of software applications both at product and at class levels of granularity, with sufficient discretion power, while they also provided preliminary evidence for its ability to discriminate between vulnerable and clean software components and hence to be used as the basis for vulnerability prediction. 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, whereas it is the only model that was built and evaluated on such a large volume of empirical data (i.e., 250 real-world software applications, comprising approximately 20 million lines of code).

Several directions for future work can be identified. First of all, we are planning to extend our work in software security assessment in order to support other programming languages including C/C++ and Python. This can be easily achieved by integrating the appropriate language-specific static analysis and software metrics tools, and by applying the model construction approach described in Section 3 in order to produce similar security assessment models. More specifically, the construction of a SAM for a new programming language includes the following steps: (i) definition of the model structure (i.e., security characteristics, properties, and measures), (ii) construction of a benchmark repository and calculation of the model thresholds, and (iii) calculation of the model weights. Within the context of the SDK4ED project a SAM for C/C++ has already been developed, based on the described approach, and has already been used by industrial partners, namely Neurasmus[26] and AIRBUS Defence and Space[27]. A detailed description of this model is available online (Online, 2020).

Currently, the proposed SAM uses the CKJM Extended tool for computing the software metrics that quantify its metric-based properties. The CKJM Extended tool operates on the bytecode-level, which restricts the practicality of the produced model, as it requires the source code to be compiled before being analyzed, an often tedious process. In the future, in order to further enhance the practicality of the proposed model, we are planning to update the proposed model by replacing the CKJM Extended tool with other open-source static analysis tools that operate directly on the source code of the software products, like JHawk[28] and OpenStaticAnalyzer[29]. We are also planning to extend the QATCH platform, by integrating the aforementioned static code analyzers.

In addition to this, the current version of the model focuses on three security characteristics, i.e., requirements, namely *Confidentiality*, *Integrity*, and *Availability*. Although these characteristics are considered the main security requirements of information systems (NIST, 2018; ISO, 2013), often termed as the CIA triad (Andress, 2014), additional requirements could be considered, such as*Authenticity* and *Non-repudiation* (ISO, 2011), in order to enhance the completeness of the model by providing more fine-grained assessments and also cover special cases of software that exhibit specific security needs. Hence, in the future, we are planning to extend our model by including additional security characteristics. In order to do this though, there would be a need for tools that are able to statically detect issues that are related to these security characteristics, whereas there should be also information available on well-accepted security knowledge bases that will enable the derivation of a reliable set of weights.

For the quantification of the alert-based properties, a single static code analyzer was utilized both for the SAM that operates on Java projects (presented in Section 3), and for the similar model that operates on C/C++ projects (described in the *Appendix* section of this paper). As already mentioned, this decision was made in order to avoid increasing the complexity of the models, as well as for preserving their practicality. However, the inclusion of additional static code analyzers would allow the model to detect new types of security issues (i.e., vulnerabilities) covering, in that way, the security needs of more types of software projects. Hence, a possible direction for future work would be to extend these models by integrating multiple static analysis tools. Nevertheless, in order to avoid affecting the practicality of the produced models due to the resulting aggregation of the alerts of multiple analyzers, appropriate techniques for filtering out those alerts that are unactionable should be investigated and integrated (Muske & Serebrenik, 2016; Heckman & Williams, 2013).

Appendix

The main goal of the present paper was to examine whether state-of-the-art concepts from the field of software quality evaluation can be leveraged (and potentially extended) for building models able to reliably assess software security. To this end, a SAM for software

[26] http://www.neurasmus.com/

[27] https://www.airbus.com/space.html

[28] http://www.virtualmachinery.com/jhawkprod.htm

[29] https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer

products written in Java programming language was carefully constructed and used as a proof-of-concept through elaborate experimentation. The internal structure of the model, along with the steps that were followed for its construction, was described in detail in Section 3 in order to help the reader (i.e., researchers or practitioners) build similar models that better meet their needs.

Within the context of the SDK4ED Project, we have also built a similar model for assessing the security level of C/C++ software products. This model was built in order to satisfy the needs of two of the project's use case providers, namely Neurasmus[30] and AIRBUS Defence and Space[31], who wanted to use the proposed security assessment concepts for assessing the security level of embedded software applications written in C/C++ that run on implantable devices and drones respectively. For building the new model, the steps presented in Section 3 were followed. In the rest of this section, the internal characteristics of the new model are presented. We believe that the information provided in this section will further facilitate interested researchers and practitioners in building similar models. It should be noted that the similar security assessment model for C/C++ software products is also available online on the web site with the supporting material of the paper (Online, 2020) as a standalone offline tool, whereas it is also integrated in the SDK4ED Platform as a web service.

Static analysis tools selection

Initially, the QATCH platform was extended by integrating appropriate static code analyzers for C/C++ programs. More specifically, the CCCC[32] tool was integrated for allowing the calculation of software metrics. The CCCC tool is a static code analyzer that operates on the sources of software programs written in C/C++, in order to compute popular software metrics, including those provided by the CK Metric suite (Chidamber & Kemerer, 1994). The CCCC metrics tool is also necessary for calculating the *SAVDs* of the defined vulnerability categories (see Section 3.1.3), since it allows the calculation of the lines of code of the analyzed software applications.

For the alert-based properties, the Cppcheck[33] static code analyzer was utilized. Cppcheck is a static analysis tool that allows the identification of software bugs in C/C++ software applications. The tool is able to detect a large number of security issues (i.e., potential vulnerabilities), including buffer overflows, memory leaks, race conditions, null pointer dereferences, usage of dangerous string functions (e.g., strcpy()), etc. It is listed by NIST[34] in their list of static analysis tools recommended for security auditing purposes.

One advantage of Cppcheck over its counterparts is that it contains rules (i.e., checkers) that verify the compliance of the source code with guidelines provided by the CERT-C standard (Seacord, 2008). Table 21 presents some indicative examples of the mapping between the Cppcheck checkers and the CERT-C guidelines. In addition, there are also checkers that evaluate the compliance of the source code with the MISRA C 2012 standard (Bagnara, 2018). However, due to licensing agreement, detailed information about the mapping between the tool checkers and the MISRA C guidelines is not freely accessible

[30] http://www.neurasmus.com/

[31] https://www.airbus.com/space.html

[32] https://sarnold.github.io/cccc/

[33] http://cppcheck.sourceforge.net/

[34] https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

Table 21 Mapping between Cppcheck checkers and CERT-C guidelines – Representative Examples

Checker	CERT-C Guideline
arrayIndexOutOfBounds	ARR30-C. Do not form or use out-of-bounds pointers or array subscripts
arrayIndexThenCheck	ARR30-C. Do not form or use out-of-bounds pointers or array subscripts
nullPointer	EXP34-C. Do not dereference null pointers
nullPointerDefaultArg	EXP34-C. Do not dereference null pointers
outOfBounds	ARR30-C. Do not form or use out-of-bounds pointers or array subscripts
possibleBufferAccessOutOfBounds	ARR30-C. Do not form or use out-of-bounds pointers or array subscripts
ignoredReturnValue	EXP12-C. Do not ignore values returned by functions
leakReturnValNotUsed	MEM31-C. Free dynamically allocated memory when no longer needed

by the users. Finally, a build-in mapping between the tool checkers and the CWE entries is also available by the Cppcheck tool. This is important since it facilitates the derivation of a reliable set of weights through the approach presented in Section 3.2.2 of the paper.

Definition of the model properties

The model should be able to analyze software applications that are written both in C and in C++ programming languages. However, although the two languages are similar regarding

Table 22 The alert-based properties (i.e., vulnerability categories) of the proposed security assessment model. The most representative CWE entry that better describes each property is also provided

Vulnerability Category	Description
Overflow	Contains bugs that may potentially lead to buffer overflows (e.g. improper array bounds checking).
I/O Issues	Contains bugs that are relevant to the dangerous usage of the I/O functionalities of the C/C++ language (e.g. bad usage of sprint, invalid usage of output stream etc.).
Exception Handling	Contains bugs that are relevant to improper or incorrect handling of exceptions or errors.
Resource Handling	Contains bugs that are relevant to improper management of system resources (e.g. memory leaks, race conditions, etc.).
Strng Issues	Contains bugs that are relevant to the misusage of C-style strings (e.g. usage of the unsafe strcpy()).
Null Pointer	Contains bugs that are relevant to Null Pointer deference.
Misused Functionality	Contains rules that check for misused functions that are provided by the programming language or widely used APIs
Dead Code	Contains issues that are relevant to the existence of unreachable code, unused variables and functions.
Arithmetic Issues	Contains bugs that are relevant to arithmetic operations and expressions. (e.g. division by zero, improper type casting, improper variable initialization, dangerous sign conversion, etc.)

Table 23 The final thresholds of the alternative security assessment model that is able to analyse software applications written in C/C++.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
t_l	0	0	0	0	0	0	0	0	0
t_m	1	4.8	0.1	0.3	2.9	0.3	0.1	1.1	0.1
t_u	8.4	23.4	1	3.2	22.5	3.2	1.5	13.9	1

P_1 Arithmetic Issues; P_2 Dead Code; P_3 Exception Handling; P_4 I/O Issues; P_5 Misused Functionality; P_6 Null Pointer; P_7 Overflow; P_8 Resource Handling; P_9 String Issues

their syntax, they belong to different programming paradigms. In particular, C++ is an object-oriented (OO) programming language, whereas C is just a structural programming language. This means that the OO software metrics cannot be computed for the case of programs written in C. Hence, in order to avoid potential problems caused by the applicability (or non-applicability) of software metrics, and since software metrics are considered security indicators, but of weak strength (see Section 3.2.2 of the paper) (Shin & Williams, 2008b; Chowdhury & Zulkernine, 2011; Shin et al., 2011; Medeiros et al., 2017; Siavvas et al., 2017b; Moshtari et al., 2013; Moshtari & Sami, 2016; Stuckman et al., 2017; Ferenc et al., 2019; Jimenez et al., 2019; Zhang et al., 2019), the produced security assessment model was decided to be based exclusively on alert-based properties.

For the definition of the alert-based properties of the new security model, the checkers of the Cppcheck were manually inspected and grouped into nine vulnerability categories based on their relevance. The alert-based properties (i.e., vulnerability categories) that were defined through this process are presented in Table 22.

Thresholds and weights derivation

For the calculation of the model's thresholds, a benchmark repository of real-world C/C++ software applications was required. For this purpose, a code base of C/C++ software applications was constructed by downloading open-source projects from the online GitHub repository. More specifically, 100 software applications (half of them written in C and the other half written in C++) were downloaded based on their popularity (i.e., GitHub stars) and analyzed using the selected static analysis tools. More specifically, the CCCC and Cppcheck tools were employed, in order to calculate the *SAVDs* of the vulnerability categories

Table 24 The final weights of the characteristics of the alternative security assessment model that is able to analyse software applications written in C/C++ as derived from the AHP (Saaty, 2008) approach. The weights reflect both the impacts retrieved from the CWE knowledge base, and the expert judgments

Property Name	Confidentiality	Integrity	Availability
Arithmetic Issues	0.1226	0.1643	0.0981
Dead Code	0.0646	0.0282	0.223
Exception Handling	0.1413	0.0908	0.0653
I/O Issues	0.1887	0.0516	0.1216
Misused Functionalities	0.0606	0.2645	0.1507
Null Pointer	0.1063	0.0783	0.0504
Overflow	0.1404	0.1092	0.0302
Resource Handling	0.1225	0.1311	0.1836
String Issues	0.0443	0.0906	0.0772

presented in Table 22. Subsequently, the thresholds of the vulnerability categories were computed based on the equations presented in Section 3.2.1 of the paper. The final computed thresholds are presented in Table 23.

The weights of the model were determined based on the knowledge retrieved from the CWE Knowledge Base, utilizing the approach presented in Section 3.2.2 of the paper. The build-in mapping between the Cppcheckers and the CWE entries, facilitated the weights elicitation procedure. The mapping is available on the web site with the supporting material of the present paper (Online, 2020). The final weights of the model are presented in Table 24.

Acknowledgements This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through SDK4ED project under Grant Agreement No. 780572.

Funding Information This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through SDK4ED project under Grant Agreement No. 780572.

Data Availability Statement The supporting material and the data that were used or produced by the present study are available at (Online, 2020).

Declarations

Conflicts of interest There are no conflicts of interest or competing interests to report.

References

Abdulrazeg, A. A., Norwawi, N. M., & Basir, N. (2012). Security metrics to improve misuse case model. *2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic*, pages 94–99.

Alhazmi, O. H., Malaiya, Y. K., & Ray, I. (2007). Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers and Security, 26*(3), 219–228.

Alshammari, B., Fidge, C., & Corney, D. (2010). Security Metrics for Object-Oriented Designs. In *2010 21st Australian Software Engineering Conference*, pages 55–64.

Alshammari, B., Fidge, C., & Corney, D. (2011). A Hierarchical Security Assessment Model for Object-Oriented Programs. *2011 11th International Conference on Quality Software*, pages 218–227.

Alshammari, B., Fidgeand, C., & Corney, D. (2009). Security metrics for object-oriented class designs. *Proceedings - International Conference on Quality Software,* 11–20.

Andress, J. (2014). *The basics of information security : understanding the fundamentals of InfoSec in theory and practice*. Waltham, MA: Syngress.

Ansar, S. A., Alka, & Khan, R. A. (2018). A phase-wise review of software security metrics. In *Networking Communication and Data Knowledge Engineering*.

Baggen, R., Correia, J. P., Schill, K., & Visser, J. (2012). Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal, 20*(2), 287–307.

Bagnara, R., Bagnara, A., & Hill, P. M. (2018). The misra c coding standard and its role in the development and analysis of safety-and security-critical embedded software. In *International Static Analysis Symposium*, pages 5–23. Springer.

Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., & Gyimóthy, T. (2011). A probabilistic software quality model. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 243–252. IEEE.

Bansiya, J., & Davis, C. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering, 28*(1), 4–17.

Basso, T., Silva, H., & Moraes, R. (2019). *On the use of quality models to characterize trustworthiness properties*. In Software Engineering for Resilient Systems: Springer.

Bholanath, R. (2016). 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)*, 1:470–481.

Boland, T. & Black, P. E. (2012). Juliet 1.1 C/C++ and java test suite. *Computer*, 45(10):88–90.

Carvalho, M., DeMott, J., Ford, R., & Wheeler, D. A. (2014). Heartbleed 101. *IEEE Security Privacy, 12*(4), 63–67.

Chess, B., & McGraw, G. (2004). Static analysis for security. *Security & Privacy, IEEE, 2,* 76–79.

Chidamber, S. R., & Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering, 20*(6), 476–493.

Chowdhury, I., Chan, B., & Zulkernine, M. (2008). Security metrics for source code structures. *Proceedings of the fourth international workshop on Software engineering for secure systems - SESS '08*.

Chowdhury, I. & Zulkernine, M. (2010). Can Complexity, Coupling, and Cohesion Metrics Be Used As Early Indicators of Vulnerabilities? In *Proceedings of the 2010 ACM Symposium on Applied Comp.*

Chowdhury, I., & Zulkernine, M. (2011). Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems,*. Architecture.

Cohen, J. (2013). *Statistical power analysis for the behavioral sciences*. Academic press.

Colombo, R. T., Pessôa, M. S., Guerra, A. C., Filho, A. B., & Gomes, C. C. (2012). Prioritization of software security intangible attributes. *ACM SIGSOFT Software Engineering Notes, 37*(6), 1.

Cunningham, W. (1993). The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger, 4*(2), 29–30.

Dam, H. K., Tran, T., Pham, T. T. M., Ng, S. W., Grundy, J., & Ghose, A. (2018). Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering,*.

Dayanandan, U. & Kalimuthu, V. (2018). Software architectural quality assessment model for security analysis using fuzzy analytical hierarchy process (fahp) method. *3D Research*, 9(3):31.

Deissenboeck, F., Juergens, E., Lochmann, K., & Wagner, S. (2009). Software quality models: Purposes, usage scenarios and requirements. In *Proc - International Conference on Software Engineering*.

DeMarco, T. (1986). *Controlling Software Projects: Management, Measurement, and Estimates*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

di Biase, M., Rastogi, A., Bruntink, M., & van Deursen, A. (2019). The delta maintainability model: Measuring maintainability of fine-grained code changes. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*.

Dromey, R. G. (1995). A model for software product quality. *IEEE Transactions on Software Engineering,21*(2), 146–162.

Edwards, W., & Barron, F. (1994). SMARTS and SMARTER: Improved Simple Methods for Multiattribute Utility Measurement. *Organizational Behavior and Human Decision Processes, 60*(3), 306–325.

Felderer, M., Büchler, M., Johns, M., Brucker, A. D., Breu, R., & Pretschner, A. (2016). Security testing: A survey. In *Advances in Computers*, volume 101, pages 1–51. Elsevier.

Ferenc, R., Hegedűs, P., Gyimesi, P., Antal, G., Bán, D., & Gyimóthy, T. (2019). Challenging machine learning algorithms in predicting vulnerable javascript functions. In *Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*.

Goseva-Popstojanova, K., & Perhinschi, A. (2015). On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology, 68,* 18–33.

Hahn, A., Tamimi, A., & Anderson, D. (2018). Securing your ics software with the attacksurface host analyzer (aha). In *Proceedings of the 4th Annual Industrial Control System Security Workshop*.

Hatzivasilis, G., Papaefstathiou, I., & Manifavas, C. (2016). Software security, privacy, and dependability: Metrics and measurement. *IEEE Software,33*(4),

Heckman, S. & Williams, L. (2009). A model building process for identifying actionable static analysis alerts. In *Proceedings - 2nd International Conference on Software Testing, Verification, and Validation, ICST 2009*, pages 161–170.

Heckman, S. & Williams, L. (2013). A Comparative Evaluation of Static Analysis Actionable Alert Identification Techniques. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, pages 4:1–4:10.

Heitlager, I., Kuipers, T., & Visser, J. (2007). A Practical Model for Measuring Maintainability. *6th International Conference on the Quality of Information and Communications Technology*.

Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. *ACM SIGPLAN Notices, 39*(12), 92.

Howard, M. (2003). *Writing secure code*. Redmond, Wash: Microsoft Press.

Howard, M. & Corporation, M. (2007). *Determining Relative Attack Surface*. US patent 7299497 B2, Patent and Trademark Office.

Howard, M., LeBlanc, D., & Viega, J. (2010). *24 Deadly Sins of Software Security*. McGraw-Hill.

Howard, M. & Lipner, S. (2006). *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press.

Howard, M., Pincus, J., & Wing, J. M. (2005). Measuring relative attack surfaces. In *Computer Security in the 21st Century*, pages 109–137. Springer.

ISO/IEC. (2011). *ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. ISO/IEC.

ISO/IEC. (2013). *ISO/IEC 27001:2013(en) Information technology Security techniques Information security management systems Requirements*. ISO/IEC.

Izurieta, C., & Prouty, M. (2019). Leveraging secdevops to tackle the technical debt associated with cybersecurity attack tactics. In *Proc. of the 2nd International Conference on Technical Debt*.

Izurieta, C., Rice, D., Kimball, K., & Valentien, T. (2018). A position study to investigate technical debt associated with security weaknesses. In *2018 International Conference on Technical Debt*.

Jankovic, M., Kehagias, D., Siavvas, M., Tsoukalas, D., & Chatzigeorgiou, A. (2019). The SDK4ED Approach to Software Quality Optimization and Interplay Calculation. In *15th China-Europe International Symposium on Software Engineering Education*.

Jimenez, M., Papadakis, M., & Le Traon, Y. (2016). Vulnerability prediction models: A case study on the linux kernel. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–10. IEEE.

Jimenez, M., Rwemalika, R., Papadakis, M., Sarro, F., Le Traon, Y., & Harman, M. (2019). The importance of accounting for real-world labelling when predicting software vulnerabilities. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

Jin, C., & Jin, S. W. (2014). Software reliability prediction model based on support vector regression with improved estimation of distribution algorithms. *Applied Software Computing, 15,* 113–120.

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681.

Kalouptsoglou, I., Siavvas, M., Tsoukalas, D., & Kehagias, D. (2020). Cross-project vulnerability prediction based on software metrics and deep learning. In *International Conference on Computational Science and Its Applications*, pages 877–893. Springer.

Kehagias, D., Jankovic, M., Siavvas, M., & Gelenbe, E. (2021). Investigating the interaction between energy consumption, quality of service, reliability, security, and maintainability of computer systems and networks. *SN Computer Science, 2*(1), 1–6.

Khurshid, S., Shrivastava, A. K., & Iqbal, J. (2019). Effort based software reliability model with fault reduction factor, change point and imperfect debugging. *International Journal of Information Technology*.

Krsul, I. (1998). *Software Vulnerability Analysis*. PhD thesis, Department of Computer Sciences, Purdue University.

Lai, S. T. (2010). An analyzer-based software security measurement model for enhancing software system security. *Proceedings - 2010 2nd WRI World Congress on Software Engineering*.

Li, B., Zhang, Y., Li, J., Yang, W., & Gu, D. (2018). Appspear: Automating the hidden-code extraction and reassembling of packed android malware. *Journal of Systems and Software, 140,* 3–16.

Luszcz, J. (2018). Apache struts 2: how technical and development gaps caused the equifax breach. *Network Security, 2018*(1), 5–8.

Manadhata, P. K., & Wing, J. M. (2011). An attack surface metric. *IEEE Transactions on Software Engineering, 37*(3),

McGraw, G. (2006). *Software Security: Building Security In*. Addison-Wesley Professional.

McGraw, G. (2008). Automated code review tools for security. *Computer, 41*(12), 108–111.

Medeiros, N., Ivaki, N., Costa, P., & Vieira, M. (2017). Software metrics as indicators of security vulnerabilities. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 216–227. IEEE.

Medeiros, N., Ivaki, N., Costa, P., & Vieira, M. (2018). An approach for trustworthiness benchmarking using software metrics. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 84–93.

Medeiros, N. P. D. S., Ivaki, N. R., Costa, P. N. D., & Vieira, M. P. A. (2017). Towards an approach for trustworthiness assessment of software as a service. In *2017 IEEE International Conference on Edge Computing (EDGE)*, pages 220–223.

Mohammed, N. M., Niazi, M., Alshayeb, M., & Mahmood, S. (2016). *Exploring Software Security Approaches in Software Development Lifecycle: A Systematic Mapping Study*. Comp: Stand. & Interf.

Morrison, P., Moye, D., Pandita, R., & Williams, L. (2018). Mapping the field of software life cycle security metrics. *Information and Software Technology, 102*(May), 146–159.

Moshtari, S., & Sami, A. (2016). 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*, pages 1415–1421.

Moshtari, S., Sami, A., & Azimi, M. (2013). Using complexity metrics to improve software security. *Computer Fraud and Security, 2013*(5), 8–17.

Mumtaz, H., Alshayeb, M., Mahmood, S., & Niazi, M. (2018). An empirical study to improve software security through the application of code refactoring. *Information and Software Technology*, 96.

Munaiah, N., Camilo, F., Wigham, W., Meneely, A., & Nagappan, M. (2017). Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project. *Empirical Software Engineering, 22*(3),

Munaiah, N., & Meneely, A. (2016). Beyond the Attack Surface: Assessing Security Risk with Random Walks on Call Graphs. *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 3–14.

Muske, T., & Serebrenik, A. (2016). Survey of Approaches for Handling Static Analysis Alarms. *in 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166.

NIST. (2018). *SP 800-160: Systems Security Engineering Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems*. National Institute of Standards and Technology.

Nunes, P., Medeiros, I., Fonseca, J., Neves, N., Correia, M., & Vieira, M. (2019). An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. *Computing*.

Online (Last Accessed 04/27/2020). Supporting Material. https://sites.google.com/view/sec-model-supp

Rindell, K., Bernsmed, K., & Jaatun, M. G. (2019). Managing security in software: Or: How i learned to stop worrying and manage the security technical debt. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19.

Rindell, K., & Holvitie, J. (2019). Security risk assessment and management as technical debt. In *International Workshop on Secure Software Engineering in DevOps and Agile Development*.

Roumani, Y., Nwankpa, J. K., & Roumani, Y. F. (2016). Examining the relationship between firm financial records and security vulnerabilities. *International Journal of Information Management, 36*(6), 987–994.

Ruthruff, J., Penix, J., Morgenthaler, J., Elbaum, S., & Rothermel, G. (2008). Predicting accurate and actionable static analysis warnings. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 341–350. IEEE.

Saaty, T. L. (2008). Decision making with the analytic hierarchy process. *International Journal of Services Sciences,*.

Scandariato, R., Walden, J., Hovsepyan, A., & Joosen, W. (2014). Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering, 40*(10), 993–1006.

Seacord, R. C. (2008). *The CERT C secure coding standard*. Pearson Education.

Sentilles, S., Papatheocharous, E., and Ciccozzi, F. (2018). What do we know about software security evaluation? A preliminary study. In *6th International Workshop on Quantitative Approaches to Software Quality*.

Shin, W., Lee, J., Park, D., & Chang, C. (2014). Design of authenticity evaluation metric for android applications. In *2014 Fourth International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pages 275–278.

Shin, Y., Meneely, A., Williams, L., & Osborne, J. A. (2011). Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering, 37*(6), 772–787.

Shin, Y., & Williams, L. (2008a). Is Complexity Really the Enemy of Software Security. *, Proc. the 4th ACM Workshop on Quality of Protection, Alexandria, Virginia, USA, Oct.*

Shin, Y., & Williams, L. A. (2008b). An empirical model to predict security vulnerabilities using code complexity metrics. In *2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*.

Siavvas, M., Chatzidimitriou, K., & Symeonidis, A. (2017a). QATCH - An adaptive framework for software product quality assessment. *Expert Systems with,*. Applications.

Siavvas, M., Gelenbe, E., Kehagias, D., & Tzovaras, D. (2018a). Static analysis-based approaches for secure software development. In *International ISCIS Security Workshop*, pages 142–157. Springer.

Siavvas, M., Jankovic, M., Kehagias, D., & Tzovaras, D. (2018b). Is Popularity an Indicator of Software Security? In *2018 IEEE 9th International Conference on Intelligent Systems (IS)*.

Siavvas, M., Kehagias, D., & Tzovaras, D. (2017b). A preliminary study on the relationship among software metrics and specific vulnerability types. *International Conference on Computational Science and Computational Intelligence (CSCI), 2017,* 916–921.

Siavvas, M., Marantos, C., Papadopoulos, L., Kehagias, D., Soudris, D., & Tzovaras, D. (2019a). On the Relationship between Software Security and Energy Consumption. In *Proceedings of the 15th China-Europe International Symposium on Software Engineering Education*.

Siavvas, M., Tsoukalas, D., Jankovic, M., Kehagias, D., & Tzovaras, D. (2020a). Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises. *Enterprise Information Systems,* 1–43.

Siavvas, M., Tsoukalas, D., Jankovic, M., Kehagias, D., Tzovaras, D., Anicic, N., & Gelenbe, E. (2019b). An empirical evaluation of the relationship between technical debt and software security. In *9th International Conference on Information Society and Technology*.

Siavvas, M., Tsoukalas, D., Marantos, C., Tsintzira, A. A., Jankovic, M., Soudris, D., Chatzigeorgiou, A., & Kehagias, D. (2020b). The sdk4ed platform for embedded software quality improvement-preliminary overview. In *International Conference on Computational Science and Its Applications*, pages 1035–1050. Springer.

Spearman, C. (1987). The proof and measurement of association between two things. By C. Spearman, 1904. *The American journal of psychology*, 100(3-4):441–471.

Stuckman, J., Walden, J., & Scandariato, R. (2017). The effect of dimensionality reduction on software vulnerability prediction models. *IEEE Transactions on Reliability, 66*(1), 17–37.

Sultana, K. Z., Deo, A., & Williams, B. J. (2017). Correlation analysis among java nano-patterns and software vulnerabilities. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pages 69–76. IEEE.

Sultana, K. Z., Williams, B. J., & Bhowmik, T. (2019). A study examining relationships between micro patterns and security vulnerabilities. *Software Quality Journal, 27*(1), 5–41.

Tang, Y., Zhao, F., Yang, Y., Lu, H., Zhou, Y., & Xu, B. (2015). 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*, pages 27–36.

Theisen, C., Munaiah, N., Al-Zyoud, M., Carver, J. C., Meneely, A., & Williams, L. (2018). Attack surface definitions: A systematic literature review. *Information and Software Technology*.

Vale, G., Fernandes, E., & Figueiredo, E. (2019). On the proposal and evaluation of a benchmark-based threshold derivation method. *Software Quality Journal, 27*(1), 275–306.

Verendel, V. (2009). Quantified security is a weak hypothesis. *Proceedings of the 2009 workshop on New security paradigms workshop - NSPW '09*, page 37.

Wagner, S. (2013). *Software product quality control*. Springer.

Wagner, S., Goeb, A., Heinemann, L., Kläs, M., Lampasona, C., Lochmann, K., et al. (2015). Operationalised product quality models and assessment: The Quamoco approach. *Information and Software Technology, 62,* 101–123.

Wagner, S., Lochmann, K., Heinemann, L., Klas, M., Trendowicz, A., Plosch, R., Seidi, A., Goeb, A., & Streit, J. (2012). The Quamoco product quality modelling and assessment approach. *2012 34th International Conference on Software Engineering (ICSE)*, pages 1133–1142.

Walden, J. & Doyle, M. (2012). SAVI: Static-Analysis vulnerability indicator. *IEEE Security and Privacy*.

Walden, J., Doyle, M., Welch, G. A., & Whelan, M. (2009). Security of open source web applications. *3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*.

Whitman, M. E., & Mattord, H. J. (2011). *Principles of information security*. Cengage Learning.

Wolff, E. (2016). *Microservices: flexible software architecture*. Addison-Wesley.

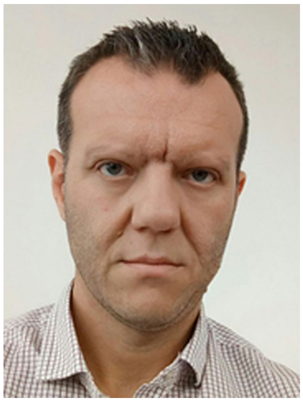
Xu, H., Heijmans, J., & Visser, J. (2013). A practical model for rating software security. *Proceedings - 7th International Conference on Software Security and Reliability Companion, SERE-C 2013*.

Zafar, S., Mehboob, M., Naveed, A., & Malik, B. (2015). Security quality model: an extension of Dromey's model. *Software Quality Journal, 23*(1),

Zhang, M., & de Carnde Carnavalet, X., Wang, L., & Ragab, A., (2019). Large-scale empirical study of important features indicative of discovered vulnerabilities to assess application security. *IEEE Transactions on Information Forensics and Security, 14*(9), 2315–2330.

Miltiadis Siavvas received the Diploma degree in electrical and computer engineering from the Aristotle University of Thessaloniki in 2016, and the Ph.D. degree in Software Security and Reliability from the Intelligent Systems and Networks Group, Imperial College London, in 2019. He is currently a Research Associate at the Information Technologies Institute of the Centre for Research and Technology Hellas (CERTH/ITI). His main research interests lie in the areas of software engineering (with emphasis on software quality, reliability, and security), machine learning and data science.

Dionysios Kehagias received the Diploma and Ph.D. degrees in electrical and computer engineering from the Aristotle University of Thessaloniki, Thessaloniki, Greece, in 1999 and 2006, respectively. He is currently a Researcher Grade C with the Information Technologies Institute of the Centre for Research and Technology Hellas (CERTH). His research interests include software technologies, algorithms, data mining and machine learning time-series analysis, big data analytics, service-oriented architectures and ontology-based knowledge engineering.



Dimitrios Tzovaras received the Diploma and Ph.D. degrees in electrical and computer engineering from the Aristotle University of Thessaloniki, Thessaloniki, Greece, in 1992 and 1997, respectively. He is currently the Director with the Information Technologies Institute of the Centre for Research and Technology Hellas. His main research interests include software engineering, visual analytics, 3-D object recognition, search and retrieval, behavioral biometrics, assistive technologies, information and knowledge management, multimodal interfaces, computer graphics, and virtual reality.

Erol Gelenbe received the B.S. degree from Middle East Technical University, Ankara, Turkey, the M.S. and Ph.D. degrees in electrical engineering from the Polytechnic Institute of New York University, Brooklyn, NY, USA, and the Doctorat d Étatès Sciences Mathématiques degree from the Université Pierre et Marie Curie, Paris, France. He is a Professor with the Institute of Theoretical and Applied Informatics, Polish Academy of Sciences (IITIS-PAN), Warsaw, Poland, and a Visiting Professor with the Imperial College, London, U.K. He retired from the Dennis Gabor Professorship at the Imperial College at the beginning of 2019, and from the Professorship at Université Paris-Descartes, in 2005. Renowned for developing mathematical models of computer systems and networks, inventing G-Networks and random neural networks, he has graduated 83 Ph.D. students, including 25 in the U.K. He has been a recipient of honorary doctorates from the Universita di Roma II, Italy (1996), Bogazic¸i University, Istanbul, Turkey (2004), and Université de Liége, Belgium (2006); he was the recipient of the Parlar Foundation Science Award (1994), the Grand Prix France Telécom of the French Academy of Sciences (1996), the ACM-SIGMETRICS Life-Time Achievement Award (2008), the IET Oliver Lodge Medal (2010), the "In Memoriam Dennis Gabor Prize" of the Hungarian Academy of Sciences (2013), and the Mustafa Prize (2017). Further, he was the recipient of the Knight of the Legion of Honour (2014) and Officer of Merit (2002) of France; and the Commander of Merit (2005) and Grand Officer of the Order of the Star (2007) of Italy, for services to higher education and research. He is on the editorial board of the IEEE TRANSACTIONS ON CLOUD COMPUTING and other journals, and is the Editor-in-Chief of the Springer journals—the Nature and the Computer Science. He is a Fellow of Academia Europaea (2005), the French National Academy of Technologies (2008), the Science Academy of Turkey (2012); he is a Foreign Fellow of the Royal Academy of Belgium (2015) and the Hungarian (2010) and Polish (2013) Academies of Science. He is also a Fellow of the ACM, the Royal Statistical Society, and the Institution for Engineering and Technology.

Authors and Affiliations

Miltiadis Siavvas[1,2] · **Dionysios Kehagias**[2] · **Dimitrios Tzovaras**[2] · **Erol Gelenbe**[1,3]

Dionysios Kehagias
diok@iti.gr

Dimitrios Tzovaras
dimitrios.tzovaras@iti.gr

Erol Gelenbe
erol.gelenbe@imperial.ac.uk

1 Imperial College London, SW7 2AZ, London, UK

2 Centre for Research and Technology Hellas, Thessaloniki, Greece

3 Institute of Theoretical & Applied Informatics, Polish Academy of Sciences, ul. Baltycka 5, 44100 Gliwice, Poland