Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises

Miltiadis Siavvas , Dimitrios Tsoukalas , Marija Jankovic , Dionysios Kehagias & Dimitrios Tzovaras

Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises

Miltiadis Siavvas[a], Dimitrios Tsoukalas[a,b], Marija Jankovic[a], Dionysios Kehagias[a] and Dimitrios Tzovaras[a]

[a]Centre for Research and Technology Hellas, Information Technologies Institute, Thessaloniki, Greece; [b]Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

ABSTRACT

Vulnerability prediction facilitates the development of secure software, as it enables the identification and mitigation of security risks early enough in the software development lifecycle. Although several factors have been studied for their ability to indicate software security risk, very limited attention has been given to technical debt (TD), despite its potential relevance to software security. To this end, in the present study, we investigate the ability of common TD indicators to indicate security risks in software products, both at project-level and at class-level of granularity. Our findings suggest that TD indicators may potentially act as security indicators as well.

ARTICLE HISTORY

Received 27 January 2020
Accepted 11 September 2020

KEYWORDS

Software engineering; technical debt; software security; vulnerability prediction; decision making

1. Introduction

A software vulnerability is a weakness in the specification, development, or configuration of software such that its exploitation can violate a security policy (Krsul 1998). The exploitation of a single vulnerability may lead to far-reaching consequences to the owing enterprise of the compromised software, including financial losses and reputation damages. For instance, Equifax Breach (CVE-2017-5638[1]) (Luszcz 2018) allowed criminals to expose the personal data of more than 143 million Equifax customers, leading to a total cost of $1.35 billion according to the company's financial results of the first quarter of 2019[2]. Hence, in order to avoid potential damages, software development enterprises are seeking mechanisms able to assist them in identifying and removing vulnerabilities as early in the development cycle as possible.

Vulnerability prediction is a technique that enables the early detection of security risks in the software development lifecycle (SDLC) (Siavvas et al. 2018a). Research endeavours in vulnerability prediction focus primarily on analysing the ability of particular software-related factors (e.g. software metrics) to detect vulnerabilities in software, as well as on developing vulnerability prediction models based on these factors, e.g. (Shin et al. 2011; Scandariato et al. 2014; Dam et al. 2018). Vulnerability prediction facilitates decision making during the SDLC, leading to the production of more secure software.

CONTACT Miltiadis Siavvas siavvasm@iti.gr Centre for Research and Technology Hellas, Information Technologies Institute, Thessaloniki, Greece

One interesting software-related factor that may indicate software security risks is Technical Debt (TD) (Cunningham 1993). TD, a notion inspired by the financial debt, is utilised as a quality metric. More specifically, it is used to quantify long-term software quality problems that are caused by quality compromises that provide short-term benefits. In fact, it is used to quantify the effort that is required for fixing design and code quality issues (i.e. code smells and violations of coding rules and best practices), which are introduced by the developers due to sacrifices they make to the quality of the code they produce usually in an attempt to meet strict production deadlines. As a result, an increased value of TD indicates that the corresponding software product contains an increased number of quality issues, which, in turn, indicates poor overall quality.

Recently, several researchers have started theoretically examining the feasibility of using TD as an indicator of security risk (Rindell, Bernsmed, and Jaatun 2019; Rindell and Holvitie 2019; Izurieta et al. 2018; Izurieta and Prouty 2019). Since most of the software vulnerabilities are caused by coding and design errors (McGraw 2006; Chess and McGraw 2004; Howard, LeBlanc, and Viega 2010), it is reasonable to expect TD indicators to also indicate security issues. However, although a multitude of highly diverse software-related factors have been empirically examined for their ability to indicate software security risk, including software metrics (Shin and Williams 2008a, 2008b; Siavvas, Kehagias, and Tzovaras 2017), text features (Neuhaus et al. 2007; Scandariato et al. 2014; Pang, Xue, and Wang 2017; Dam et al. 2018), product popularity (Siavvas et al. 2018b) or even firm's financial records (Roumani, Nwankpa, and Roumani 2016), very limited attention has been given on TD. In fact, although various software metrics (e.g. CK Metrics (Chidamber and Kemerer 1994)), which are occasionally treated as quality indicators, have been widely studied for their relevance to software security, actual low-level TD indicators (e.g. bugs, code smells, duplicated code, etc.), as well as actual high-level TD indicators (e.g. SQALE Index) have not been studied yet. The only known attempt can be found in a study by Siavvas et al. (2019), in which we empirically evaluated the relationship between SQALE Index and software security risk, showing that a statistically significant relationship may exist.

To this end, in the present paper, based on the preliminary findings of our previous work (Siavvas et al. 2019), we investigate the ability of common TD indicators to indicate software security risk. More specifically, in this study, we examine the predictive performance of TD indicators in predicting software security risks both at project-level and at class-level of granularity, by building different machine learning (ML) models. As far as project-level analysis is concerned, we examine the ability of TD indicators to predict the security risk level of a software project, based on the results of static analysis. To do so, we initially constructed a large repository comprising 210 open-source Java applications retrieved from GitHub,[3] which were analysed using a popular static analysis platform in order to calculate their TD indicators, as well as their security risk, measured in terms of vulnerability density. The security risk was then discretised in security risk levels based on a set of data-driven thresholds in order to be suitable for the construction of the classification models. Based on the produced dataset, several ML models were built considering both the cases of binary (i.e. two security risk levels) and 3-class (i.e. three security risk levels) classification, in order to cover different enterprise needs. The

produced ML models were evaluated based on popular performance metrics, which were also used for selecting the best model in each case.

Subsequently, emphasis was given on class-level of granularity, and particularly on the ability of various carefully selected TD indicators to predict the existence of actual vulnerabilities in software classes. For this purpose, a highly balanced vulnerability dataset was constructed based on OWASP Benchmark,[4] which is a popular dataset of vulnerable and clean classes. Based on this dataset, both correlation and discriminant analysis were conducted with the purpose to detect potential relationships between the studied TD indicators and the existence of vulnerabilities, whereas the performance of these indicators in class-level vulnerability prediction was examined through the construction of ML models.

The results of the project-level analysis revealed that TD indicators could potentially predict the security risk level of a software project with sufficient accuracy, with the Logistic Regression algorithm to be the best performing model. On the other hand, the class-level analysis highlighted the capacity of TD indicators to discriminate between vulnerable and clean software classes, and, in turn, the feasibility of constructing relatively accurate class-level vulnerability prediction models based on these indicators, with Random Forest demonstrating the best predictive performance. The results of our study suggest that TD indicators may indicate software security risks, and therefore they could potentially be used as part of a security assessment process.

The rest of the paper is structured as follows. Section 2 provides a state-of-the-art analysis focusing on the open issues that the present work attempts to address. In Section 3, the project-level analysis is described in detail, whereas Section 4 is dedicated to the class-level analysis. In Section 5 we present the validity threats of the present paper and how we are trying to mitigate them. In Section 6 the implications of the present work to researchers and practitioners are discussed. Finally, Section 7 concludes the paper and discusses directions for future work, whereas a discussion of how the present paper is related to the Software Development Enterprises is also provided.

2. Related work

There is a multitude of research attempts in the field of software security aiming at identifying security risk indicators. In fact, specific emphasis has been given in the related literature on the ability of software-related factors to predict the existence of potential vulnerabilities in software products or components, which are identified either from actual vulnerability reports (e.g. Shin et al. 2011; Chowdhury and Zulkernine 2011), or through static analysis (e.g. Scandariato et al. 2014; Pang, Xue, and Wang 2017; Dam et al. 2018).

More specifically, the capacity of common software metrics to indicate the existence of software vulnerabilities has been extensively studied. The first attempts were made by Shin and Williams (2008a, 2008b) and Chowdhury and Zulkernine (2010) who observed that common complexity, coupling, and cohesion (CCC) metrics may have a significant (albeit weak) relationship to the existence of vulnerabilities in software components. Based on this observation, they also investigated the predictive performance of these metrics, i.e. their performance in predicting the existence of vulnerabilities in software

components, which was found to be promising. The ability of software metrics to be used as vulnerability indicators (i.e. predictors) has been supported by numerous follow-up studies (e.g. Moshtari and Sami 2016; Siavvas, Kehagias, and Tzovaras 2017; Ferenc et al. 2019; Jimenez et al. 2019). Along with CCC metrics, these studies investigated additional software metrics.

Significant attention has been given also on information retrieved directly from source code, either through static analysis or through text mining. Regarding static analysis, Gegick et al. (2008) observed that a close correlation may exist between the static analysis alerts density and the actual vulnerabilities that a program contains. This observation was supported by the results of relatively recent empirical studies, e.g. (Walden, Stuckman, and Scandariato 2014; Yang, Ryu, and Baik 2016). As far as text mining is concerned, the first known attempt was made by Neuhaus et al. (2007), who examined the ability of tokens (i.e. keywords directly retrieved from the product source code) to indicate the presence of vulnerabilities in software artefacts. A notable attempt was made by Scandariato et al. (2014), who managed to built text mining-based vulnerability predictors with sufficient predictive performance. Recently, there is a shift in the related literature towards adopting deep learning in an attempt to improve the predictive performance of text mining-based vulnerability predictors, e.g. (Pang, Xue, and Wang 2017; Dam et al. 2018).

Factors non-directly related to the product source code have recently started gaining the attention of the research community. For example, Roumani, Nwankpa, and Roumani (2016) found a strong correlation between the financial records of the software development enterprises (e.g. sales, financial performance, etc.) and the number of vulnerabilities that their products may contain. Recently, Siavvas et al. (2018b) investigated whether open-source software products' popularity can be used as an indicator of their security level, concluding that popularity may not constitute a reliable security indicator.

Despite the multitude of research endeavours focusing on the ability of highly diverse factors to indicate security risks in software products, almost no research attempts exist explicitly focusing on dedicated TD indicators, such as code smells and code duplication. Software metrics are the only indicators indirectly related to TD that have been studied so far, which have demonstrated only weak correlation to software security. However, studying only the indirect indicators is not sufficient for generalising these findings to the TD itself. Recently, several researchers have started examining the feasibility of quantifying software security indirectly through the notion of TD. More specifically, Rindell et al. (Rindell, Bernsmed, and Jaatun 2019; Rindell and Holvitie 2019) provided guidelines on how the concept of TD can be extended to support software security, whereas Izurieta et al. (Izurieta et al. 2018; Izurieta and Prouty 2019) presented ways for prioritising security bugs as TD items (i.e. quality issues). However, these studies provide only theoretical evaluation of the feasibility of TD to be used as a security indicator, without providing empirical evidence for the relationship between TD and software security. The only known attempt can be found in (Siavvas et al. 2019). In this study, we empirically evaluated the relationship between TD and software security based on a repository of 50 open-source software products, showing that a statistically significant and strong

correlation exists between these two factors. This work provides preliminary evidence for the inter-relationship between TD and software security risk.

In the present paper, we extend our previous work with the purpose to reach safer conclusions regarding the potential relationship between TD and software security. More specifically, the project-level analysis of our present work is based on a much larger code base comprising 210 open-source software applications. In addition, contrary to our previous work in which we focused exclusively on TD Principal (i.e. SQALE Index), in this study we consider 12 TD indicators, providing in that way finer-grained analysis. Moreover, in the previous study the analysis was focused exclusively on project-level of granularity. In the present study, we also examine the ability of TD indicators to indicate security risks at lower levels of granularity and specifically on class-level. In addition, contrary to previous studies in which only some indicators with indirect relevance to TD were considered (i.e. software metrics), in the present work previously uninvestigated factors with direct relation to TD are studied including code smells, bugs, and code duplication. Finally, in our previous work, we focused only on the correlation between the values of TD and software security (i.e. vulnerability density). On the contrary, in the present study, we focus on the predictive performance of TD indicators, i.e. on their ability to predict security risk levels (i.e. project-level analysis) or actual vulnerabilities in software classes (i.e. class-level analysis), leading to the construction of actual predictors. To the best of our knowledge, this is the first study that specifically investigates the predictive performance of dedicated TD indicators, such as code smells, bugs, and code duplication, in indicating software security risk.

3. Project-level analysis

In the present section we investigate the ability of TD indicators to indicate security risks in software projects at project-level of granularity. More specifically, we examine whether TD indicators can be used as the basis for predicting the security risk level of software products, measured in terms of the security-related static analysis alerts density. This information would be very useful both for developers and project managers, since poor TD would also indicate questionable security, and therefore, the accumulation of TD liabilities would indicate possibly the accumulation of security-related issues (i.e. potential vulnerabilities), allowing them to make early decisions about code testing and refactoring.

3.1. Experiment setup and methodology

3.1.1. Overview of the methodology

In Figure 1, a high-level overview of the overall approach that is adopted in the present paper for examining the ability of common TD indicators to predict software security risk at project-level of granularity is illustrated. As can be seen in Figure 1, the overall approach comprises five steps, which are briefly described below:

1. **Data Definition**. The first step of the study is to define the structure of the dataset that will be used for the construction of the ML models. This involves the selection of the input variables (i.e. TD indicators), as well as of the class attribute (i.e. Security Risk Levels) of the predictors. Regarding the input variables, apart from common software metrics (e.g. complexity), numerous previously uninvestigated TD indicators were considered for their ability

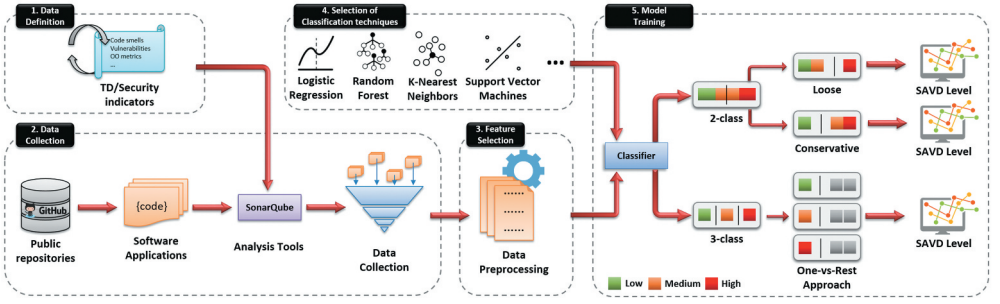

Figure 1. High-level overview of the overall model construction approach for the project-level analysis.

to predict software security. The class attribute was defined by discretising the values of a popular security risk indicator into three security risk levels, namely Low, Medium, and High.

2. **Data Collection**. The purpose of this step is to retrieve a sufficient number of real-world open-source software products and to analyse them using a popular static analysis platform in order to calculate their TD indicators and their software security risk levels, leading to the construction of the dataset defined in the previous step.

3. **Data Pre-processing**. This step is responsible for cleaning the data and bringing the dataset in a form ready to be used for training purposes. More specifically, this step involves all the required pre-processing actions, including feature selection, data resampling, and hyper-parameter tuning.

4. **Selection of Classification Techniques**. In this step, the most suitable classification techniques are selected, taking into account the specific characteristics of the dataset that is produced by the previous steps of the approach.

5. **Model Training**. This final step of the overall approach is responsible for building and evaluating classification models using the dataset that was constructed in the previous steps. More specifically, it is responsible for building classifiers able to classify a given software application into the correct security risk level, based on the values of its TD indicators. Proper model evaluation techniques are applied in order to ensure that the best classifier is selected in each case.

It should be noted that in the final step we consider both the cases of binary (i.e. 2-class) and 3-class classification. In the case of the 3-class classification, which we term *General* approach, the produced models focus on assessing whether the security risk level of a given application is Low, Medium or High according to the values of its TD indicators and previous knowledge (i.e. retrieved from popular open-source repositories). These models provide fine-grained security assessments, allowing developers and project managers to track the security risk level of their applications throughout their overall SDLC.

In the case of the binary classification, the three security risk levels are reduced into two, by merging the Medium category either with the Low or with the High risk level accordingly. Hence, two approaches are considered, namely (i) a *Conservative* approach, in which we built models that are able to predict the Low | Medium+High classes, and (ii) a *Loose* approach, in which we build models that are able to predict the Low+Medium |

High classes. The binary models are useful for developers and project managers that are not interested in having fine-grained security assessment, but instead to be notified when the security risk of their applications is expected to exceed a specific threshold in order to act promptly. The *Conservative* approach is more suitable for software development enterprises that built security-critical applications, as in these enterprises the security risk should be minimised as much as possible. Other enterprises may opt for the *Loose* approach, in order to reduce the burden caused by frequent notifications by the models.

The above analysis suggests that the proposed approach can be tailored to satisfy the needs of different enterprises. A more detailed description of the steps presented above is provided in the rest of this section.

3.1.2. Selected indicators

3.1.2.1. Technical debt. The majority of TD indicators proposed in the literature discover TD items that are linked with software aspects (Tsoukalas et al. 2018; Li, Avgeriou, and Liang 2015; Alves et al. 2016), which enables the evaluation of different software artefacts' characteristics. With respect to object-oriented software, object-oriented structural properties (Chidamber and Kemerer 1994; Li and Henry 1993), such as the widely known complexity, coupling, and cohesion, have been widely utilised for predicting the maintenance effort and, in turn, the maintainability of software (Riaz, Mendes, and Tempero 2009; Van Koten and Gray 2006; Fioravanti and Nesi 2001; Zhou and Leung 2007; Zhou and Xu 2008), a quality attribute that is closely related to TD. For instance, with respect to the property of complexity, various studies have addressed the impact of Cyclomatic Complexity, i.e. the number of linearly independent paths through a program's source code, as a predictor of the maintainability of a software project (Giger, Pinzger, and Gall 2012; Bruntink and van Deursen 2006; Singh and Saha 2012). In a similar manner, coupling metrics, such as the Coupling Between Objects (CBO) or the Coupling Between Methods (CBM), and cohesion metrics, such as the Lack of Cohesion in Methods (LCOM), have been considered by a multitude of researchers for their ability to measure and predict (Eski and Buzluca 2011; Shatnawi and Wei 2008; Zhou et al. 2012; Zhou and Leung 2007; Van Koten and Gray 2006; Shatnawi and Wei 2008; Elish and Elish 2009). Therefore, since TD is closely related to the maintainability quality attribute, the aforementioned software metrics are usually being treated as indirect TD indicators or a subset of TD indicators (Kosti et al. 2017; Tsoukalas et al. 2020; Alves et al. 2016; Siebra et al. 2014).

Besides OO metrics, assessment tools used in the industry that employ well-known models, such as the ISO/IEC 25022–25023 standard or the SQALE (Letouzey and Ilkiewicz 2012) methodology, all gather their atomic data by calculating various software factors (i.e. measures). Some of these measures upon which assessment tools estimate code-level TD are code duplication, and code coverage, among others. There exist various studies that relate each of these metrics with TD (Griffith et al. 2014; Nugroho, Visser, and Kuipers 2011; Marinescu 2012). In addition, there are various studies that explore the involvement of code smells in the presence of TD (Alves et al. 2016; Palomba et al. 2018). Code smells can be described as code or design patterns that often violate one, or more than one programming principle (Fowler 2018), thus leading to deeper problems in further development and maintenance of the software. These problems may impede the software

maintenance process and impose the need for code refactoring. Finally, various static code analysers manage to identify TD through source code analysis, which aims in locating bugs or violations that can cause quality decay (Xuan, Hu, and Jiang 2017; Griffith et al. 2014; Digkas et al. 2017). These code-level issues are normally eliminated through the application of code refactoring (Zazworka et al. 2013).

SonarQube[5] is the world's leading static analysis platform for continuous inspection of code quality that provides analysis functionalities and a wide range of metrics for measuring quality attributes of code, tests, and design. As of today, it has been adopted by more than 120 K organisations including nearly more than 100 K public open-source projects.[6] In this study, SonarQube has been used as proof of concept for research purposes, since according to two recent studies on TD Management (Li, Avgeriou, and Liang 2015; Ampatzoglou et al. 2015), it is the most frequently used tool for estimating TD principal. To do so, SonarQube checks code compliance against a set of classified coding rules and if the code violates any of these rules, it considers it a violation or a TD item. For each of the identified TD items, SonarQube computes the remediation time (i.e. estimated effort) needed to refactor it and considers it as TD. Therefore, in the present work, we opted for the TD-related metrics (computed both on the project- and class-level of granularity) that are provided by SonarQube, as our primary TD indicators to predict software security risk. In fact, SonarQube has been used to statistically analyse the selected 210 software applications (see Section), as well as the 2740 software classes of the OWASP Benchmark (see Section 4.2). A more detailed definition of the selected metrics is provided in Section.

3.1.2.2. Security.

3.1.2.2.1. Software security risk score. Similarly to our previous study (Siavvas et al. 2019), we chose the Static Analysis Vulnerability Density (SAVD) (Walden et al. 2009) metric as our main software security indicator. In fact, the Vulnerability Density metric is 'the total number of vulnerabilities that a software product contains per thousand lines of code' (Alhazmi, Malaiya, and Ray 2007). The vulnerability density metric is quantified based either on the number of vulnerabilities reported on vulnerability databases (Shin and Williams 2008a; Chowdhury and Zulkernine 2010) or based on the results of static analysis (Walden and Doyle 2012; Siavvas, Kehagias, and Tzovaras 2017; Siavvas et al. 2019). The SAVD is actually the Vulnerability Density metric that is computed based on the security-related results of static analysis tools (i.e. vulnerabilities identified through static analysis). The SAVD metric is a popular security risk indicator widely used in the related literature (Walden et al. 2009; Walden and Doyle 2012; Siavvas, Kehagias, and Tzovaras 2017; Siavvas et al. 2018b; Walden and Doyle 2012), whereas it has been also extensively used for vulnerability prediction purposes (Gegick et al. 2008; Yang, Ryu, and Baik 2016).

For quantifying the SAVD of the selected software applications, similarly to the case of TD indicators, we used SonarQube in order to extract security-relevant issues (i.e. potential vulnerabilities). In fact, SonarQube has been used for calculating the SAVD of the selected 210 software applications (see Section). To detect potential vulnerabilities, SonarQube uses various popular analysers (such as FindBugs[7] and PMD[8]) under the hood and aggregates their reports, giving additional value by incorporating also its own technologies and custom security rules tailored to each supported programming

language. These security-related rules are based on well-established security-standards, such as CWE, SANS, and OWASP. In fact, SonarQube, by making use of these rules, it is capable of detecting important security issues that reside in the source code, and also tries to reduce the number of the produced false positives to the greatest possible extent.[9]

In the context of the present experiment, to compute the SAVD of each project, we divided the total number of potential vulnerabilities identified by SonarQube for that software product with the lines of code of the project, also computed by SonarQube. The result was multiplied by 1000, to express SAVD per thousand lines of code. For better understanding, the SAVD of a given software application is given by the following formula:

$$SAVD = 1000\frac{N_{vuln}}{LOC} \tag{1}$$

where:

N_{vuln}: The total number of potential vulnerabilities that the software application contains as reported by SonarQube

LOC: The total Lines of Code of the software application as reported by SonarQube

The SAVD value denotes how many potential vulnerabilities the software application contains per thousand lines of code according to SonarQube. As a result, the higher the SAVD of a software application, the higher its security risk, as it contains more (on average) security-related static analysis alerts (i.e. security issues), which are likely to manifest themselves as vulnerabilities. Hence, in the present study, the value of SAVD is used as the value of the *Security Risk Score (SRS)* of a given software application. It should be noted that the normalisation of the metric by the LOC of the software product is necessary, for making the measure independent of the product size, and therefore able to be used for the direct comparison of different software products.

3.1.2.2.2. Software security risk levels. During software development, developers and project managers are interested in knowing the security risk level of their software applications in order to better plan their testing and refactoring activities. In fact, they would like to be notified when the security risk level of their application is high in order to act promptly. Although SAVD is a sufficient indicator of software security risk, due to the fact that its value is numerical, it does not provide information about whether its value can be considered high or low. Hence, a discrete indicator of software security risk is necessary to facilitate decision making. In fact, a discrete security risk indicator is more sufficient for communicating assessment results even to non-technical stakeholders (e.g. managers) since the human brain can better perceive linguistic values compared to actual numbers.

For this purpose, we decided to create a discrete security risk indicator by discretising the SAVD into different *Security Risk Levels (SRLs)*. In order to achieve this, a set of thresholds needs to be defined. We decided to use three thresholds for the discretisation of the SAVD, namely t_l, t_m, t_u, which correspond to the lower, middle, and upper threshold of the SAVD respectively. The values of these thresholds could be determined either based on expert judgements, or based on real-world data. The latter approach was selected in order to avoid the subjectivity that underlies expert judgements and therefore leads to a more reliable set of thresholds. Among the existing threshold derivation approaches, we decided to use the benchmarking approach since it is the most widely used in the related

literature, e.g. (Heitlager, Kuipers, and Visser 2007; Wagner et al. 2012; Siavvas, Chatzidimitriou, and Symeonidis 2017).

The benchmark repository of 210 open-source software applications (described in Section 3.1.3) based on which the ML models of the present study are constructed was used for the calculation of these thresholds. More specifically, SonarQube was applied to each one of the applications of the selected repository and their SAVDs were computed by applying Equation (1). Subsequently, the three thresholds were calculated based on the distribution of the SAVDs, by applying the following formulas, which were proposed by Wagner et al. (2012):

$$\begin{aligned} t_l &= \min(\{s : s \geq Q_{25\%}(s_1, \ldots, s_n) - 1.5 \cdot IRQ(s_1, \ldots, s_n)\}) \\ t_m &= \text{median}(s_1, \ldots, s_n) \\ t_u &= \max(\{s : s \leq Q_{75\%}(s_1, \ldots, s_n) + 1.5 \cdot IRQ(s_1, \ldots, s_n)\}) \end{aligned} \tag{2}$$

where:

s_i denotes the SAVD value of the i-th product of the benchmark repository

Q_p denotes the p-percentile

n denotes the size of the selected benchmark repository

$IRQ(s_1, \ldots, s_n)$ denotes the inter-quartile-range

In simple words, the minimum, median, and maximum values of SAVD were selected as its lower, middle, and upper thresholds respectively (after removing the outliers). The final thresholds and the SRLs that were produced from the above procedure are illustrated in Table 1.

From Table 1 it is clear that, since the lower threshold has a value of 0, three SRLs are defined. A software application is assigned to one of these SRLs based on the value of SAVD as determined by SonarQube. This discretisation enables the construction of classification models able to predict the risk level of a software application based on its TD Indicators, which will be presented in the rest of this section. It should be noted that, although here we present three SRLs, as already mentioned, binary classification is also feasible, by merging two risk levels into one (see Section 3.1.1). As stated previously, binary classification is useful in case that the developers and project managers just want to be notified when the security risk of their applications is above a specific threshold.

Table 1. The *Security Risk Levels (SRLs)* along with their corresponding values of SAVD. The thresholds of the SAVD that determine each *SRL* were computed based on a benchmark repository of 210 open-source software applications that were retrieved from GitHub.

Security Risk Level	SAVD Range
Low	$[0, 0.515)$
Medium	$[0.515, 4.068)$
High	$[4.068, \text{inf})$

At this point, it should be noted that the *Security Risk Score (SRS)* is a relative security indicator, which actually compares a given software product to other real-world software products that are available on the market with respect to 'how well it avoids security issues (reported through static analysis)'. A High *SRS* (in fact, SRL) suggests that the project contains much more security-relevant static analysis alerts (i.e. security issues) on average compared to a baseline of real-world projects that are available on the market, and therefore, it is more likely to contain an actual vulnerability than the others.

3.1.3. Dataset

For the execution of this study, we aimed at combining different TD and security-related indicators into a common dataset with the purpose of investigating if and to what extent TD can be used in order to accurately predict the security risk level of a software application. To start the dataset construction process, we initially selected 210 popular open-source applications from the GitHub repository. The selected 210 applications have different sizes and belong to different application domains. The selection criteria were based on software popularity, activity level, data availability, and the Java programming language. A sufficiently large number of applications are fundamental to reach a conclusion that does not depend on a specific dataset, allowing to generalise the obtained results.

3.1.3.1. Data definition. After fetching the source code (i.e. latest commit) of each application, we proceeded to the next step, i.e. using SonarQube to perform static analysis on each codebase in order to retrieve both, TD and security indicators described in the previous Section 3.1.2. In Table 2, the 12 metrics that were selected as (direct and indirect) TD indicators and therefore used as the initial independent variables set are presented along with a short description.

Our final dataset comprises a table with 210 rows (the number of analysed applications) and 13 columns, where each one of the first 12 columns holds the value of a specific TD indicator, while an extra column at the end of the table holds the value (class) of the SRL (i.e. SAVD level). Since the purpose of the project-level analysis is to investigate the ability of TD indicators to act as security risk level predictors of a software application, the columns that refer to TD metrics will play the role of independent variables, while the last column that refers to SRL (i.e. SAVD level) will play the role of the dependent variable, i.e.

Table 2. The initial set of technical debt (TD) indicators considered by the present analysis.

Metric	Description
bugs	Total number of bug issues of a project.
code_smells	Total number of code smell issues of a project.
comment_lines	Total number of lines that correspond to comments.
open_issues	Total count of issues in the Open state.
ncloc	Total number of lines that are not part of a comment.
uncovered_lines	Total number of code lines that are not covered by unit tests.
duplicated_blocks	Total number of lines that belong to duplicated blocks.
complexity	Total Cyclomatic Complexity calculated based on the number of paths through the code.
sqale_index	Total effort (measured in terms of minutes) to fix all the identified issues.
classes	Total number of classes.
functions	Total number of functions.
cognitive_complexity	Total Cognitive Complexity, measuring the degree to which the code's control flow is understandable.

the security risk level that we try to predict. This format helped us during the classification model construction phase described in Section 3.2.

3.1.3.2. Descriptive statistics. In order to provide insights regarding the selected dataset, its descriptive statistics need to be presented. These statistics (e.g. mean, median, standard deviation, variance, minimum and maximum values, etc.) provide simple summaries about the sample and about the observations that have been made. After analysing and extracting the metrics of each application, we present the descriptive statistics of our dataset in Table 3.

Metrics that vary little are not likely to be useful predictors. In our case, from Table 3 we observe that for all metrics there are significant differences between the lower 25th (lower) percentile, the median, and the 75th (upper) percentile, thus showing strong variations. Therefore, all metrics were selected to be used for subsequent analysis.

In Figure 2, we present a histogram of the distribution of SRL (i.e. SAVD level) class among the 210 applications of our dataset. As we can see, most software applications fall under the Low SAVD level class (105 instances). In the second place, there is the Medium class (83 instances), whereas the High class is the class with the least occurrences (22 instances). This observation implies that our dataset is imbalanced, especially when it comes to the High SAVD level class, where the total count of software application instances is considerably lower than the other two classes. To investigate if this issue will affect the performance of the examined classifiers, i.e. make a classifier biased towards Low or Medium classes, we will employ resampling techniques and compare the results with those obtained without data resampling. More details on this are presented in Section.

3.1.4. Data pre-processing and model construction

3.1.4.1. Classification models. The purpose of the project-level analysis is to examine the feasibility of using classification for predicting the SRL of software applications based on TD indicators. To do so, we applied an arsenal of ML classification models and compared their results in order to select the best model. We decided to omit Artificial Neural Network models from our experiments, as the size of our dataset (210 instances) does not suffice to train such models. Below, the selected models are briefly described:

Table 3. Descriptive statistics of TD indicators of the 210 open-source real-world software applications used for the project-level analysis.

Metric	Mean value	Standard deviation	Min value	Lower quartile	Median value	Upper quartile	Max value	Skewness	Kurtosis
bugs	29.665	61.148	0	3	9	30	585	5.286	37.601
code_smells	858.660	2095.092	2	94	235	732	16442	5.551	35.262
comment_lines	5113.780	15587.257	1	387	1381	3889	170698	8.203	77.661
open_issues	927.144	2235.457	3	102	264	777	18625	5.631	36.678
ncloc	19524.278	37253.222	175	2655	7068	20438	357664	5.262	38.091
uncovered_lines	9815.507	17856.102	32	1302	3297	10288	137326	3.905	19.212
duplicated_blocks	110.177	429.327	0	0	12	47	4219	7.771	67.183
complexity	3914.024	7420.959	11	421	1352	4284	61862	4.275	23.544
sqale_index	9143.732	24074.250	10	951	2755	7540	250449	7.027	59.265
classes	250.172	529.702	1	40	94	281	6453	8.391	92.196
functions	1933.144	4033.357	7	248	725	2053	45852	7.234	70.777
cognitive_complexity	3244.833	7100.924	5	271	859	3201	59109	4.667	26.822

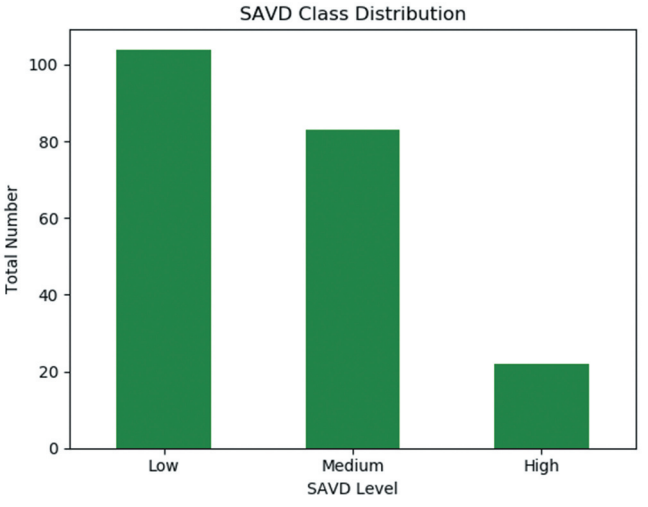

Figure 2. SAVD class distribution.

• **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 (see Section 3.2.3 for more information).

• **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 maximises 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 predict 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.

For the conduction of our experiments, we used the Python programming language and more specifically the scikit-learn[10] ML library that contains the implementation of all the above algorithms.

The scale of the input data is known to affect some of the models that we investigated, namely the KNN and SVM algorithms. To address this issue, we applied standardisation to the dataset before experimenting with these algorithms.

3.1.4.2. Feature selection. The selection of independent (i.e. input) variables before designing or experimenting with an ML algorithm is a task that needs to be treated with special attention. A large number of input features, i.e. a high-dimensional feature space, may lead to the 'curse of dimensionality' (Bellman 2003). According to this phenomenon, the increasing number of the model's inputs leads to a degradation in its predictive performance. Features that are not associated (or they are partially associated) with the class attribute can also negatively affect model performance. Thus, after constructing our dataset, the next step is to provide a clear understanding of the statistical attributes of our variables, and then to apply feature selection techniques for reducing the number of the model's inputs by keeping only the TD indicators (described in Section) that are highly significant for SRL prediction.

In order to study the statistical significance of each TD indicator over the security quality, we applied four different feature selection methods. More specifically, we used two filter-based methods, namely Spearman Correlation and Chi-Squared, one wrapper-based method named Recursive Feature Elimination (RFE), and finally one embedded method named Tree-based Elimination (TBE). In general, filter-based methods try to filter the features based on some metrics, while wrapper-based methods consider the selection of a set of features as a search problem. Embedded methods use algorithms that have built-in feature selection methods (e.g. Lasso and Random Forest). More details on the four different feature selection methods used for keeping only the TD indicators that are highly significant for SRL prediction are provided below.

- The **Spearman correlation** is a non-parametric technique used to measure the monotonicity of the relationship between the values of two datasets. Similarly to other correlation techniques (e.g. Pearson), Spearman correlation coefficients vary between −1 and +1. However, as a non-parametric test, Spearman correlation does not assume any distribution for the studied data. A coefficient of −1 or +1 implies an exact monotonic relationship, while a coefficient of 0 implies no correlation at all. In our case, we examine the absolute values of the Spearman correlation coefficients between the independent (TD Indicators) and dependent (SRL) variables in our dataset, and we rank the former based on these values. Then, we keep the top *N* features based on this criterion.

- The **Chi-Squared** method is a statistical hypothesis test used to determine whether there is a statistically significant relationship between a non-negative feature and a class, by calculating the chi-square statistic. A small chi-square test statistic means that there is a relationship between the studied variables, while a large chi-square test statistic means there is no relationship. Since the Chi-Squared method measures the dependency between a feature and a class, it can be used to filter out the features that are most likely to be independent of this specific class and therefore, not suitable for classification. In our case, we calculate the chi-square statistic between the independent (TD Indicators) and dependent (SRL) variables in our dataset, and we rank the former based on these values. Then, we keep the top *N* features based on this criterion.

- The **Recursive Feature Elimination (RFE)** method, as its name suggests, is a feature selection technique that recursively eliminates features based on an estimator model trained on the initial set of features. Depending on the nature of the model, the importance of each

feature is determined either by a coefficient attribute (e.g. Logistic Regression) or by a feature-importance attribute (e.g. Random Forest). During this process, the least important features are recursively filtered out until the final set contains only the desired number of features. In our case, we select Logistic Regression as the estimator model. As a result, the RFE assesses the importance of each feature based on the coefficient in the decision function of the Logistic Regression object and prunes the initial set until it contains only the top *N* features in terms of importance.

• **Tree-based Elimination (TBE)** is an Embedded method, meaning that it uses models that have built-in feature selection methods to eliminate the initial features' number. In this case, contrary to the RFE method, we use the Random Forest model to calculate feature importance based on node impurities in each decision tree. Then, we keep the top *N* features based on the average of all feature importance values calculated by each decision tree.

We applied each method described above independently on the feature set and retained the top $N = 5$ features that were selected by each method. We set $N = 5$ mainly due to the fact that both RFE and TBE methods stopped the feature elimination process when the feature subset reached five features. As a result, selecting the top five features from each independent method allowed us to directly compare the selected features' subsets among the four methods. Then, we aggregated the results by ranking each feature based on the number of times it was selected to be in the top five of a particular method. Table 4 displays the features ranked by the number of times they were selected by each method. A value of True in a specific column indicates that the feature of that specific row has been selected to be in the top five features of the algorithm of this column.

By having a look at Table 4, we can see that *bugs* and *open issues* features are in the top five set of every feature selection method. Moreover, *code smells* and *complexity* are also high in the list since they were selected by three out of four methods. Finally, *sqale index* is selected by two out of four methods. As our final feature set, we decided to keep only the features that were selected to be within the top five set of at least two different methods. This means that features ranked below *sqale index* in the table will be excluded from the final set. To conclude, among the initial 12 features (TD indicators) under investigation, five of them were found to have statistically significant effects on SRL, by more than one feature selection algorithms. It should be noted at this point that every feature selection method described above performs a statistical test to reach a decision regarding which features to eliminate and which to retain during the selection process. As a result, the aforementioned statistically significant effects of the final set of features with respect to

Table 4. Results of the four feature selection methods.

Feature	Spearman	Chi-2	RFE	TBE	Total
bugs	**True**	**True**	**True**	**True**	**4**
open_issues	**True**	**True**	**True**	**True**	**4**
code_smells	**True**	**True**	**True**	**False**	**3**
complexity	**False**	**True**	**True**	**True**	**3**
sqale_index	**True**	**False**	**False**	**True**	**2**
functions	False	False	True	False	1
comment_lines	False	False	False	True	1
uncovered_lines	False	True	False	False	1
duplicated_blocks	True	False	False	False	1
classes	False	False	False	False	0
cognitive_complexity	False	False	False	False	0
ncloc	False	False	False	False	0

the target variable (i.e. SRL) are the outcome and the aggregation of the individual statistical tests performed by each feature selection method. Therefore, the TD indicators that were found to be the most promising SRL predictors according to our feature selection approach are *bugs, open issues, code smells, complexity*, and *sqale index*. These metrics will be considered as input to the classification models described in Section . At this point it should be noted that in our previous work (Siavvas et al. 2019), which was based only on *sqale index*, a statistically significant and strong correlation was observed between the *sqale index* and the SAVD calculated based on another static code analyser, providing more support regarding the relevance of this TD indicator to software security.

3.1.4.3. Training configuration. Once our dataset is ready for supervised learning, the next step is to train and validate the performance of the selected classification algorithms. The dataset contains a total of 210 project entries. To evaluate our model performance, we initially used the Train-Test split approach, where we randomly split the dataset into two sets: 75% for training and 25% for test. However, since our data is limited there is a possibility of high bias, because there is a high chance that the models may miss some information about the data which is not used in the training set.

To overcome this challenge and ensure that every observation from the original dataset has the chance of appearing in training and test set, we also used the K-Fold cross-validation approach (Mosteller and Tukey 1968), which generally results in a less biased model compare to other methods. Using K-Fold cross-validation will result in more models being trained, and in turn, a more accurate estimate of the performance of the models on unseen data. In this approach, we set K = 10 and the training was conducted over the entire dataset. This means that the dataset was randomly split into 10 folds, where the K-1 folds were used to train the model, while the remaining Kth fold was used to test the model. The whole process was repeated until each of the K-folds has served as the test set. We recorded the errors on each of the predictions for each of the K-Folds and computed the average of the K recorded errors (i.e. the cross-validation error) that served as our performance metric for the model.

Before the learning process begins, a hyper-parameter tuning process must take place in order to increase models' predictive performance. A model hyper-parameter is an external attribute of the model that cannot be estimated from data during the training process. In order to tune our models in the best possible way, we used GridSearchCV,[11] a Python implementation of the Grid-search method (Feurer et al. 2015). Grid-search is commonly used to find the optimal hyper-parameters of a model that result in the most accurate predictions, by performing an exhaustive search over specified parameter values for an estimator. We chose F1 score as the objective function of the estimator to evaluate a parameter setting. We performed hyper-parameter selection on every classifier under investigation during the 10-Fold cross-validation described above to avoid overfitting and ensure that the selected models have a good degree of generalisation.

Classification problems having multiple classes with imbalanced dataset generally oppose a challenge, as the skewed distribution makes many conventional ML algorithms less effective, especially in predicting minority class examples. As described in Section, our dataset is imbalanced due to the fact that it contains considerably more samples for the Low and Medium SRL classes than for the High SRL class. This can make a classifier biased towards the one or two classes, i.e. learn the classes with more samples better (Low and

Medium class) and remain weak on the smaller (High) class. In order to investigate if this issue affects the performance of our classifiers, we used the Synthetic Minority Over-sampling Technique (SMOTE) (Chawla et al. 2002) to augment the dataset with artificial data. We repeated the experiments twice, once with applying SMOTE and once without SMOTE and compared the results (see Section 3.2). It is worth mentioning that SMOTE was applied only to the training set, as over-sampling on test data imposes a bias on the findings.

3.1.4.4. Performance evaluation. We evaluated and compared the classification performance of the investigated models using Precision, Recall and F1 score metrics. Given a class C that we try to predict, Precision is the ratio of instances correctly predicted as class C, i.e. True Positives (TP), to the total number of predictions with class C, i.e. True Positives (TP) + False Positives (FP). The formula for calculating Precision is as follows:

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

Recall, on the other hand, is the ratio of instances that are correctly predicted as class C, i.e. True Positives (TP), to the total number of instances with an actual class of C, i.e. True Positives (TP) + False Negatives (FN).

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

Finally, F1 score is the harmonic mean of Precision and Recall and reaches its best value at 1 and worst at 0. The relative contribution of Precision and Recall to the F1 score is equal. The formula for the F1 score is:

$$F1score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{5}$$

3.2. Experimental results

Three experiments were made for three independent approaches: a *Loose* approach using 2-class classification for predicting the SRL between Low+Medium|High, a *Conservative* approach using 2-class classification for predicting the SRL between Low|Medium+High, and finally a *General* approach using 3-class classification for predicting the SRL between Low|Medium|High. These approaches have been described in detail in Section 3.1.1.

3.2.1. Predicting security risk level using the loose approach

As discussed in the previous section, F1 score is selected as the main evaluation metric for classification since it combines both Precision and Recall. Table 5 shows Precision, Recall and F1 cross-validation scores for all models, while the experiments were repeated two times, one using SMOTE for data resampling and one without SMOTE.

From Table 5 we can clearly see that Logistic Regression is by far the best model in terms of average Precision, Recall and F1 score. More specifically, in the case of classification performance without applying SMOTE, the F1 score of Logistic Regression is 0.863, while the second best model is XGBoost with an F1 score of 0.581. Respectively, after

Table 5. Cross-validation averaged scores for all models for loose approach.

Classifier	SMOTE	Accuracy	Precision	Recall	F1 Score
Logistic Regression	**Yes**	**0.955**	0.891	**0.908**	**0.889**
	No	0.951	**0.926**	0.847	0.863
K-NN	Yes	0.710	0.553	0.616	0.537
	No	0.880	0.498	0.514	0.501
Random Forest	Yes	0.852	0.627	0.696	0.644
	No	0.904	0.551	0.541	0.533
Gaussian Naïve Bayes	Yes	0.396	0.552	0.572	0.334
	No	0.814	0.475	0.491	0.470
XGBoost	Yes	0.818	0.595	0.691	0.612
	No	0.905	0.605	0.578	0.581
SVM(linear)	Yes	0.472	0.568	0.661	0.423
	No	0.895	0.447	0.500	0.472

applying SMOTE on the training set, the F1 score of Logistic Regression is 0.889, while the second best model is Random Forest with an F1 score of 0.644. Besides Logistic Regression, Random Forest and XGBoost, the performance scores of the rest of the models are too low to be further investigated.

Table 6 shows the confusion matrix of the best performing model, Logistic Regression. Cross-validation approach is not suitable for producing an overall confusion matrix, as different confusion matrices are produced at each evaluation run. For that reason, we used the Train-test split approach, i.e. we randomly split the dataset into two sets: 75% for training the Logistic Regression model and 25% for testing the model, and then calculated the confusion matrix. In this case, we did not use a resampling approach, as we wanted to investigate the performance of the model in dealing with data imbalance between classes.

As can be seen in the confusion matrix, there is only one misclassified instance for the Low+Medium class that was predicted as High, while the remaining 48 instances were correctly predicted as Low+Medium. For the High class, all four instances were correctly classified, indicating again that the algorithm is able to perform very well, even for the minority classes and without any resampling. Averaged Precision, Recall and F1-score for the Train-test split approach are 0.90, 0.99 and 0.94 respectively.

Among the various ML models that were built having as input the TD indicators and as output the SRLs, Logistic Regression was found to be the best model, demonstrating high predictive performance with an average F1 score of 0.889. It should be noted that regression models (including logistic regression) have demonstrated sufficient performance in vulnerability prediction in the related literature (Gegick et al. 2008; Roumani, Nwankpa, and Roumani 2016; Camilo, Meneely, and Nagappan 2015; Shin et al. 2011). The high predictive performance of Logistic Regression can be attributed mainly to two factors: i) the possibility that features (i.e. TD indicators) and the target variable (i.e. SRL) are governed by linear relationships (i.e. linear patterns) and thus, linearly separable

Table 6. Train-test split confusion matrix of logistic regression model for loose approach.

	1: Low+Medium	2: High
1: Low+Medium	**48**	1
2: High	0	**4**

decision boundaries can be easily created, and ii) the fact logistic regression work better in the absence of features that are unrelated (noise) to the target variable as well as features that are correlated to each other.

In general, logistic regression performs better when the number of unrelated variables is significantly less (or zero) compared to the number of related independent variables, while non-linear algorithms, such as Random Forest, have higher accuracy as the number of independent variables increases in a dataset. In the case of SRL prediction performed at project-level, we tried to eliminate unrelated variables and decrease the number of independent variables as much as possible, by performing and combining various feature selection techniques (see Section). This process, also known as dimensionality reduction, resulted in a significantly less-complex dataset, which allowed Logistic Regression to use its full capabilities, thus confirming in that way the fact that simplicity of linear models can occasionally lead to providing better results than other more sophisticated models, such as non-linear or ensemble learners (Kirasich, Smith, and Sadler 2018).

3.2.2. Predicting security risk level using the conservative approach

Table 7 shows cross-validation scores for all models using the Conservative Approach, while again the experiments were repeated with and without applying the SMOTE technique. We can observe that similarly to the *Loose* approach, Logistic Regression is the model achieving the highest classification scores. When experiments are conducted without applying SMOTE, Logistic Regression has an F1 score of 0.892, while the F1 score of the second best model (i.e. XGBoost) is 0.646. The application of SMOTE does not have an effect on the F1 score of the Logistic Regression whereas XGBoost comes second again, with an F1 score of 0.661. Besides Logistic Regression and XGBoost, Random Forest is performing slightly worse than XGBoost, with an F1 score of 0.635 and 0.647 respectively. Again, F1 score of the remaining models is not considered worthy of further investigation.

In Table 8 the confusion matrix of the best-performing model (i.e. the Logistic Regression model) is shown. Again, for the calculation of the confusion matrix the Train-test split approach without any data resampling was used.

While inspecting the confusion matrix, we can observe that there are only two misclassified instances for the Medium class that were predicted as Medium+High. The remaining 25 instances were correctly classified as Low. For the Medium+High class, 23 instances were correctly classified, while three instances were misclassified as Low. The

Table 7. Cross-validation averaged scores for all models for conservative approach.

Classifier	SMOTE	Accuracy	Precision	Recall	F1 Score
Logistic Regression	**Yes**	**0.893**	**0.901**	**0.894**	**0.892**
	No	0.893	0.901	0.894	0.892
K-NN	Yes	0.603	0.612	0.604	0.596
	No	0.603	0.612	0.604	0.596
Random Forest	Yes	0.651	0.657	0.651	0.647
	No	0.637	0.641	0.637	0.635
Gaussian Naïve Bayes	Yes	0.554	0.589	0.556	0.468
	No	0.554	0.589	0.556	0.468
XGBoost	Yes	0.665	0.676	0.666	0.661
	No	0.651	0.662	0.652	0.646
SVM(linear)	Yes	0.558	0.617	0.560	0.495
	No	0.558	0.617	0.560	0.495

Table 8. Train-test split confusion matrix of logistic regression model for the conservative approach.

	1: Low	2: Medium+High
1: Low	**25**	2
2: Medium+High	3	**23**

average Precision, Recall and F1-score that are computed using the Train-test split approach are 0.91, 0.91 and 0.91 respectively. For more information about why (according to the authors) Logistic Regression demonstrated better results than other ML algorithms, we refer the reader to Section 3.2.1.

3.2.3. Predicting security risk level using the general approach

Most of the selected classification algorithms presented in Section natively support multiclass problems. Logistic regression, however, is a statistical method for predicting binary classes, as the outcome of this algorithm is dichotomous in nature. To overcome this limitation of Logistic Regression to support multiclass predictions, we extended the model by applying the one-vs-rest scheme (Bishop 2006). One-vs-rest (OvR), also known as one-vs-all, is a strategy that decomposes the multiclass optimisation problem into separate binary classifiers that are trained separately for all classes, and then combines each of the classifiers' binary outputs to generate multi-class outputs.

In Table 9, cross-validation scores are given for all models, with and without data resampling. Similarly to the two approaches presented above, Logistic Regression is again the best model, since its averaged classification scores outperform those of the remaining models. This performance can be attributed to the OvR scheme that Python's algorithm implementation natively supports. In particular, SRL can be predicted with an average of 0.821 Recall, 0.858 Precision, and 0.836 F1 score for the case of classification without SMOTE. The second best model is XGBoost with an F1 score of 0.502, which is considered way too low compared to the best performing model. In the case of data sampling, SRL can be predicted with an average Recall of 0.821, Precision of 0.858 and F1 score of 0.836. Random Forest classifier comes second with an F1 score of 0.503, a score which again is considered to be too far from that of Logistic Regression. The F1 score of the remaining investigated models is below 0.5 and is therefore not worth mentioning.

Table 9. Cross-validation averaged scores for all models for general approach.

Classifier	SMOTE	Accuracy	Precision	Recall	F1 Score
Logistic Regression	Yes	0.832	0.842	**0.837**	0.814
	No	**0.836**	**0.858**	0.821	**0.817**
K-NN	Yes	0.513	0.495	0.504	0.473
	No	0.532	0.400	0.422	0.403
Random Forest	Yes	0.568	0.514	0.525	0.503
	No	0.589	0.467	0.486	0.466
Gaussian Naïve Bayes	Yes	0.485	0.305	0.355	0.296
	No	0.566	0.444	0.457	0.439
XGBoost	Yes	0.560	0.483	0.517	0.481
	No	0.590	0.501	0.521	0.502
SVM(linear)	Yes	0.322	0.413	0.463	0.302
	No	0.540	0.343	0.375	0.319

Table 10. Train-test split confusion matrix of logistic regression model for general approach.

	1: Low	2: Medium	3: High
1: Low	**26**	1	0
2: Medium	4	**16**	2
3: High	0	0	**4**

In Table 10, the confusion matrix of Logistic Regression, which is the best performing model of the general approach, is presented by using the Train-test split approach without any data resampling.

By having a look at the confusion matrix, it can be observed that the number of misclassified instances is greater for the Medium class, where out of a total of 22 samples, four instances were predicted as Low and two were classified as High. However, the other instances were classified correctly as Medium. For the Low class, only one instance was misclassified as Medium, while the rest 26 instances were classified correctly. Finally, all the instances that belong to the High class, all four instances were correctly classified. The above results indicate that the Logistic Regression algorithm is able to correctly classify a very satisfactory percentage of the instances for both, the majority and the minority classes, despite the fact that this task deals with a 3-class classification. The averaged Precision, Recall and F1-score (for the Train-split approach) were found to be 0.82, 0.90, and 0.84 respectively. For more information about why (according to the authors) Logistic Regression demonstrated better results than other ML algorithms, we refer the reader to Section 3.2.1.

4. Class-level analysis

In the previous section we focused on the project-level of granularity. More specifically, we investigated the ability of TD indicators to indicate security risks at the level of the broader software application, i.e. whether TD indicators can be used as predictors of SRLs. This analysis led us to the observation that TD metrics can potentially be used as sufficient predictors of security-related bugs (i.e. potential vulnerabilities) at project-level, and, in turn, that poor TD may also indicate questionable security. This observation is important from a practical perspective, as it can be used by project managers and developers, in order to get an indication of the security risk that is accumulated to the software applications under development due to unresolved code-level quality-related issues.

Another interesting from a practical viewpoint question that deserves individual merit (and which will also enhance the completeness of the present work) is whether the same observations hold at lower levels of granularity. In simple words, it would be worth investigating the ability of TD indicators to discriminate between vulnerable and clean software components (i.e. classes). To this end, in the present section we focus on class-level of granularity and we examine the capacity of TD indicators to predict the existence of actual vulnerabilities in software classes.

4.1. Experiment overview

Based on the description provided above, the problem is reduced to a typical problem of vulnerability prediction in software products (Siavvas et al. 2018b; Jimenez et al. 2019). The research in the field of vulnerability prediction focuses on examining the ability of several software-related factors (e.g. software metrics) to indicate the existence of vulnerabilities in software components (e.g. packages, classes, etc.), as well as on the construction of vulnerability prediction models (VPMs) based on these factors (Morrison et al. 2015). VPMs are normally built based on ML techniques that use these factors as inputs, to discriminate between vulnerable and clean software components. This information is very useful for the construction of more secure software, as knowing the existence of potentially vulnerable components will help project managers and developers better plan their testing and fortification activities by allocating limited resources to high-risk areas (i.e. security hotspots). Hence, contrary to the previous section in which we focused on predicting the SRLs of software applications, in this section we focus on identifying the existence of actual vulnerabilities in software components, and particularly software classes.

The vast majority of the research attempts in the field of vulnerability prediction follow the same procedure for investigating the ability of specific software-related factors to indicate the existence of vulnerabilities in software components. In brief, this involves three broader steps, namely: (i) the construction of a vulnerability dataset, (ii) the conduction of correlation and discriminant analysis in order to assess the relevance of the studied indicators to the existence of vulnerabilities, and (iii) the evaluation of the predictive performance of the studied indicators (i.e. their ability to predict the existence of vulnerabilities in software components) through the construction of ML models, e.g. (Chowdhury and Zulkernine 2011; Shin et al. 2011; Scandariato et al. 2014; Dam et al. 2018). In order to be in-line with the previous research endeavours we adopt a similar approach. A high-level overview of the approach that we adopt for the purposes of the class-level analysis that is presented in this section is illustrated in Figure 3.

As can be seen in Figure 3, the overall approach comprises three major steps, which are briefly described below:

1. **Dataset Construction and Preprocessing**. Similarly to the project-level analysis, the first step of our study is the construction of a dataset that will be utilised for the conduction of the correlation and discriminant analyses, and, in turn, for the construction of the ML models. Since the present analysis will focus on the ability of TD indicators to predict the existence of actual vulnerabilities in software classes, a highly balanced repository of clean and vulnerable

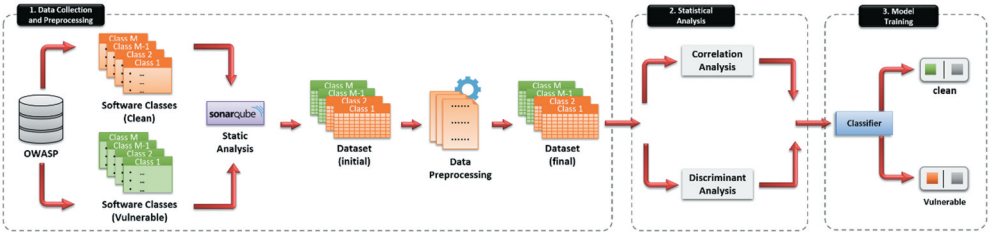

Figure 3. High-level overview of the approach followed for the class-level analysis.

software classes needs to be constructed and analysed using the selected TD analysis tool (i.e. SonarQube) (see Section). This step is also responsible for applying appropriate data cleansing techniques (in our case, duplicate removal) in order to bring the data in a form that will be usable for further analysis. The resulting dataset will form the basis of our analysis.

2. **Statistical Analysis**. As already mentioned, an important step in vulnerability prediction is to perform statistical analysis, with the purpose to examine whether a statistical relationship exists between the studied indicators and the existence of vulnerabilities in software components. In order to be in-line with the related literature, two statistical tests are applied. More specifically, *Correlation Analysis* is applied in order to examine whether the studied indicators are correlated with the existence of vulnerabilities in a statistically significant manner, whereas *Discriminant Analysis* is applied in order to examine the ability of the studied indicators to discriminate between vulnerable and clean software classes. The results of these statistical analyses play a critical role for the selection of the final set of indicators that will act as inputs of the produced ML models.

3. **Predictive Performance and Model Training**. The final step of the overall approach is responsible for building and evaluating ML models, based on the previously constructed dataset. More specifically, the main purpose is to examine the predictability of the selected TD indicators, i.e. their ability to predict the existence of actual vulnerabilities in software classes.

The results of the two final steps of the overall approach will help us assess the ability of the selected TD Indicators to indicate security risks at the level of software classes. A positive outcome of this experiment will provide further support to the findings of the previous section.

4.2. Dataset construction and preprocessing

The first step of the present analysis is the construction of a highly balanced dataset of vulnerable and clean software components. Despite the multitude of research endeavours that can be found in the field of vulnerability prediction (Shin and Williams 2008a; Chowdhury and Zulkernine 2011; Shin et al. 2011; Scandariato et al. 2014; Siavvas, Kehagias, and Tzovaras 2017; Moshtari and Sami 2016; Ferenc et al. 2019; Jimenez et al. 2019; Zhang et al. 2015), current literature lacks a balanced and reliable vulnerability dataset. In fact, the vast majority of the vulnerability datasets that are used in the literature for vulnerability prediction are manually constructed based on reported vulnerabilities. In particular, the authors manually search vulnerability databases for reported vulnerabilities and subsequently they mine online open-source repositories for retrieving the software components (i.e. packages, classes, etc.) that contain these vulnerabilities.

Although this approach leads to vulnerability datasets containing real-world software components, they are hindered by a set of important shortcomings that affect their correctness and their reliability (Siavvas et al. 2018a; Morrison et al. 2015; Jimenez et al. 2019). First of all, not all of the vulnerabilities that a product contains are always reported on online vulnerability databases, and therefore many components that are considered clean in these datasets may in fact be vulnerable, affecting in that way the correctness and reliability of the produced dataset (Morrison et al. 2015; Shin and Williams 2013). Moreover, existing datasets are highly imbalanced, e.g. (Gegick et al. 2008; Scandariato

et al. 2014; Yang, Ryu, and Baik 2016). In fact, the number of vulnerable files that a software product includes is often too small (approximately 1–5%) (Alhazmi, Malaiya, and Ray 2007), leading to highly imbalanced datasets, which influence significantly the accuracy of the produced predictors (Morrison et al. 2015; Shin and Williams 2013). Finally, the construction of these datasets is a manual process, which is inevitably prone to human errors.

Hence, for the needs of the present study, and in order to enhance the reliability of its results, we decided to follow a safer approach and use a well-accepted benchmark. In particular, a highly balanced dataset of clean and vulnerable components was constructed based on the OWASP Benchmark.[12] OWASP Benchmark is a popular test suite that is commonly used for the evaluation of static code analysers regarding their ability to detect vulnerabilities. It is a collection of a large number of software components (i.e. classes) 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 are in the desired level of granularity. Secondly, contrary to similar test suites (e.g. Juliet Test (Boland and Black 2012)), the selected benchmark comprises also software components that do not contain actual vulnerabilities (i.e. it contains clean classes). 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 2740 software classes of the OWASP Benchmark were statically analysed using SonarQube in order to compute the TD Indicators that were described in Section . It should be noted that from these indicators the *classes* indicator was not applicable in the case of class-level analysis, as the number of classes is always 1. The resulting dataset underwent a pre-processing step, with the main purpose to remove highly similar classes. More specifically, classes that received exactly the same values in the computed TD indicators were removed from the dataset as they were considered identical. After removing duplicates, the dataset consisted of 861 vulnerable and 639 clean classes. In order to construct a highly balanced dataset, 600 observations were randomly selected from each group of vulnerable and clean components. Hence, this led to the construction of a highly balanced dataset, comprising 600 vulnerable and 600 clean Java classes. The resulting dataset does not exhibit the aforementioned shortcomings of the datasets that exist in the literature as: (i) it is highly balanced, (ii) the vulnerability status of the classes (i.e. class attribute) is guaranteed to be the one stated, and (iii) it is curated by experts in the field of Software Security, and widely used as the basis for checking the ability of static and dynamic analysis tools to detect actual vulnerabilities.

A small fragment of the resulting vulnerability dataset is illustrated in Table 11. The purpose of this fragment is to demonstrate the structure of the dataset used for the present analysis. The complete dataset is available on the website with the supporting material of the present work (Online (Last Accessed 29/08/2020)).

M_1:bugs;M_2:sqale_index;M_3:code_smells;M_4:uncovered_lines;M_5: duplicated_blocks;M_6:comment_lines; M_7:ncloc;M_8:functions;M_9:complexity;M_{10}: cognitive_complexity;M_{11}:open_issues

Table 11. A fragment of the vulnerability dataset that was used for the class-level analysis, i.e. for the construction of the class-level vulnerability prediction models. The last column of the dataset denotes whether the corresponding class is vulnerable (i.e. 1), or clean (i.e. 0). The complete dataset contains 1200 software classes retrieved from the OWASP Benchmark.

Class Name	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	Vulnerability State
BenchmarkTest01734	2	178	11	43	10	8	68	3	8	8	13	1
BenchmarkTest01353	0	116	6	39	8	5	68	3	10	7	6	0
BenchmarkTest01764	1	122	5	35	9	4	61	3	6	5	6	0
BenchmarkTest01543	0	195	9	45	13	8	73	3	11	16	9	0
BenchmarkTest01993	1	150	8	52	8	3	83	3	12	15	9	0
BenchmarkTest00065	0	75	3	43	5	2	69	2	7	11	3	1
BenchmarkTest00085	1	110	9	52	4	8	78	2	11	19	9	1
BenchmarkTest00243	0	126	10	56	5	9	86	3	13	20	10	0
BenchmarkTest00859	1	133	9	64	7	7	91	2	5	9	11	1
BenchmarkTest02697	0	72	8	30	2	5	50	3	4	3	8	0
BenchmarkTest01611	0	147	9	49	9	7	79	3	13	19	9	0
BenchmarkTest01657	2	122	6	24	9	5	46	3	6	3	8	1
BenchmarkTest01815	0	92	7	25	5	7	48	3	4	4	7	0
BenchmarkTest00588	0	40	3	25	2	1	44	2	10	14	3	1
BenchmarkTest02014	0	216	14	54	10	9	84	3	13	20	14	0

As can be seen by Table 11, the rows of the dataset are the analysed classes, whereas the columns are the values of the TD Indicators as computed by SonarQube. It should be noted that the last column of Table 11 is the vulnerability state of the class, which denotes whether the corresponding class is vulnerable (i.e. 1), or clean (i.e. 0).

4.3. Statistical analysis

An important step of the present experiment is to conduct statistical analysis in order to examine whether observable relationships exist between the selected TD Indicators and the existence of vulnerabilities in the software classes of the selected benchmark. For this purpose, both correlation and discriminant analysis were applied, which are described in detail in the rest of this section. The results of these analyses are also important for the construction of the ML models presented in Section, as they can be used for selecting the inputs of the produced models.

4.3.1. Correlation analysis

In order to determine the ability of the selected TD indicators to indicate security risks in software classes, we initially applied correlation analysis, with the purpose to examine the relationship between the TD Indicators and the existence of vulnerabilities in software classes. A statistically significant relationship would provide confidence for the ability of the selected indicators to indicate the existence of vulnerabilities in software classes.

For the purposes of the present analysis, we calculated the correlation between each one of the computed TD Indicators and the Class attribute of the dataset, which is the *Vulnerability State* (see Table 11). More specifically, we decided to use the *Point-biserial correlation coefficient (r)*, which is commonly used for computing the correlation between a continuous and a dichotomous (i.e. binary) variable. For the characterisation of the correlation strength, we used the thresholds suggested by Cohen (2013). 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. However, it should be noted that we do not expect the correlations to be strong. Even a weak correlation, is normally acceptable in the literature for providing evidence for the potential ability of a factor to indicate the existence of vulnerabilities (e.g. Shin and Williams (2008a); Shin et al. (2011); Chowdhury and Zulkernine (2011); Camilo, Meneely, and Nagappan (2015); Moshtari and Sami (2016)).

The computed *Point-biserial correlation coefficients (r)* between the TD Indicators and the vulnerability states are presented in Table 12, along with their *p-values*, which are necessary for judging whether the observed correlations are statistically significant. In the present experiment, a correlation is considered to be statistically significant, if its associated *p-value* is found to be lower than the value of 0.05. To facilitate readability, the statistically significant correlations are marked with an asterisk in Table 12.

From Table 12, we can see that all the selected TD Indicators, with the only exception of *nloc* and *uncovered_lines*, demonstrate a statistically significant correlation with the *Vulnerability State* of the dataset. Another interesting observation is that almost all of the selected TD Indicators demonstrate a weak (according to Cohen (2013)) correlation with the existence of vulnerabilities. The only indicator that demonstrated a moderate correlation is the metric named *bugs*, which is also the only one that is positively correlated to the existence of vulnerabilities. The positive correlation indicates that the higher the number of bugs that a class contains, the higher the likelihood of containing an actual vulnerability. It should be noted that as stated in Section, *bugs, open_issues*, and *code_smells* contain mostly quality-related (and not security-related) issues. SonarQube reports the security-related issues as *vulnerabilities*.

The results of the correlation analysis show that the vast majority of the selected TD indicators demonstrate a statistically significant weak to moderate correlation with the existence of vulnerabilities in the classes of the selected benchmark repository. This highlights the potential ability of the selected TD indicators to indicate the existence of vulnerabilities in software classes, and therefore to be used as the basis for vulnerability prediction. However, since the observed correlations were not found to be strong, discriminant analysis is required in order to reach safer conclusions.

Table 12. The results of the correlation analysis. The asterisk denotes that the associated correlation coefficient is statistically significant in the 95% confidence interval.

TD Indicator	Correlation Coeff	*p-value*
bugs	0.4582*	$< 2.2 \times 10^{-16}$
code_smells	−0.2396*	0.1887×10^{-4}
open_issues	−0.2029*	0.0416×10^{-5}
duplicated_blocs	−0.02254*	0.1530×10^{-10}
sqale_index	−0.1172*	0.4698×10^{-7}
ncloc	−0.2241	0.4765
comment_lines	−0.2915*	0.4400×10^{-33}
uncovered_lines	−0.2238	0.06509
functions	−0.1044*	$< 2.2 \times 10^{-16}$
complexity	−0.1169*	2.26×10^{-12}
cognitive_complexity	−0.1369*	1.4×10^{-22}

4.3.2. Discriminant analysis

In the previous section we observed that most of the studied TD indicators demonstrate a statistically significant correlation to the existence of vulnerabilities in software classes. In order to reach safer conclusions regarding the relationship between TD indicators and software vulnerabilities, we also applied discriminant analysis. The purpose of discriminant analysis is to assess whether the TD indicators are able to discriminate between vulnerable and clean classes. More specifically, the purpose of this analysis is to examine whether the values of the TD indicators computed for the clean classes are significantly different in a statistical manner from the values that they receive for the vulnerable classes. If statistically significant differences are observed, this would suggest that vulnerable classes tend to receive significantly higher or lower values of TD indicators, and, in turn, that the values of these indicators could potentially be used for discriminating between vulnerable and clean classes.

As a first step of our analysis, in Table 13 we present the average values of the computed TD Indicators both for the clean and for the vulnerable classes of the analysed dataset. As can be seen in Table 13, the average values tend to be different in each TD Indicator. In all the cases, the indicators seem to receive a higher value at clean classes than in vulnerable, with the only exception of the indicator named *bugs*. This denotes that clean classes tend to contain fewer bugs than vulnerable classes in the studied dataset. Another interesting observation is that clean classes exhibit higher values in quality-related issues (e.g. *code_smells*), size metrics (e.g. *nloc*), and complexity metrics (e.g. *complexity*) than their vulnerable counterparts. This can be explained by the fact that in order to remove vulnerabilities, additional code is usually required (e.g. additional checks), which increases the size and complexity of the class, whereas quality-related issues are normally added (e.g. violations of naming conventions, etc.).

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 statistically 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 artefacts (e.g. Shin and Williams (2008b); Munaiah and Meneely (2016); Jimenez et al. (2019)). In particular, the

Table 13. The results of the discriminant analysis.

TD Indicator	Clean	Vulnerable	Wilcoxon Test (*p-value*)
bugs	0.3633	0.8167	$< 2.2 \times 10^{-16}$
code_smells	7.44	6.34	1.039×10^{-10}
open_issues	7.835	6.81	5.343×10^{-9}
duplicated_blocs	7.41	7.26	0.1099
sqale_index	125.71	116.023	0.2089×10^{-3}
ncloc	66.18	59.3	1.59×10^{-11}
comment_lines	6.965	4.92	$< 2.2 \times 10^{-16}$
uncovered_lines	40.8	35.35	4.026×10^{-12}
functions	2.7	2.615	0.01187
complexity	8.43	7.87	0.00151
cognitive_complexity	10.54	9.28	0.0001952

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 selected TD indicator of vulnerable and clean software components.*

H_1: *The values of the selected TD Indicator of the clean and vulnerable components are statistically different.*

The above test was performed for each one of the 11 TD indicators that were considered in our analysis.[13] The *p-values* of the individual tests are reported in the last column of Table 12. As can be seen by this table, the *p*-values in all the cases apart from the *duplicated_blocks* indicator were found to be lower than the threshold of 0.05. Hence, in all these cases the null hypothesis is rejected, leading to the acceptance of the alternative hypothesis. This suggests that a statistically significant difference is observed between the values of the TD indicators of clean and vulnerable classes, which indicates that these TD indicators can discriminate between vulnerable and clean classes and can potentially be used as indicators of vulnerabilities in software classes.

Finally, the results of the discriminant analysis presented in this section provide preliminary evidence for the ability of the selected TD indicators to be used as the basis for vulnerability prediction. However, a more elaborate analysis is provided in the next section.

4.4. Predictive performance and model training

In the previous section, correlation and discriminant analysis were applied with the purpose to identify whether statistical relationships are observed between the selected indicators and the existence of vulnerabilities in software classes. The results of these analyses revealed that TD indicators may also indicate the existence of vulnerabilities in software classes, as they were found to be able to discriminate between vulnerable and clean classes. In this section, building on top of the findings of Section 4.3, we focus on the predictability (i.e. predictive performance) of the selected TD indicators, i.e. their ability to predict the existence of actual vulnerabilities in software classes. To this end, several ML models are built using the selected TD indicators as inputs, and their performance in vulnerability prediction is evaluated based on well-known performance metrics.

For the purposes of the present experiment, the vulnerability dataset presented in Section 4.2 was used as the basis for the construction of the ML models. The results of the correlation and discriminant analysis were used for the selection of the subset of TD indicators that could be considered for the construction of the ML models. More specifically, the *ncloc* and *uncovered_lines* indicators were excluded from our analysis as they did not demonstrate a statistically significant correlation with the existence of vulnerabilities (see Section 4.3.1), whereas the *duplicated_blocks* indicator was also excluded, as it did not demonstrate statistically significant discriminative power (see Section 4.3.2). Based on the remaining TD indicators, a set of ML models was constructed, by applying the same set of ML algorithms that were used in the project-level analysis (see Section). Similarly to the project-level analysis, the predictive performance of the ML models was evaluated using the F1 score, based both on the train-test split and on the 10-fold cross-validation approaches. The results of the 10-fold cross-validation analysis of the produced models are illustrated in Table 14.

Table 14. Cross-validation averaged scores for all the produced class-level vulnerability prediction models.

Classifier	Accuracy	Precision	Recall	F1 Score
Logistic Regression	0.702	0.710	0.702	0.698
K-NN	0.582	0.583	0.582	0.579
Random Forest	**0.710**	**0.715**	**0.710**	**0.708**
Gaussian Naïve Bayes	0.658	0.669	0.658	0.652
XGBoost	0.700	0.705	0.700	0.698
SVM(linear)	0.704	0.710	0.704	0.702

As can be seen in Table 14, Random Forest is the best performing model with an F1 score of 0.708, followed closely by linear kernel SVM with an F1 score of 0.702. This is in-line with the findings of the vast majority of existing research endeavours in the field of vulnerability prediction (Chowdhury and Zulkernine 2011; Moshtari, Sami, and Azimi 2013; Scandariato et al. 2014; Walden, Stuckman, and Scandariato 2014; Tang et al. 2015; Zhang et al. 2015; Dam et al. 2018), in which Random Forest was also found to be the best performing ML model. Other model performances that worth mentioning here are Logistic Regression and XGBoost, with an F1 score of 0.698.

The fact that both linear and non-linear models have demonstrated sufficient and rather comparable predictive performance can be explained by the fact that obvious relationships between the selected TD indicators and the existence of vulnerabilities have been observed. More specifically, as already mentioned, all of the selected TD indicators that were used as inputs of the produced models demonstrated a statistically significant correlation with the existence of vulnerabilities (see Section 4.3.1). In addition to this, statistically significant differences in their values were observed between clean and vulnerable classes, denoting that they can discriminate between vulnerable and clean classes (see Section 4.3.2).

Table 15 shows the confusion matrix of the best performing model, i.e. Random Forest. As mentioned in the system-level analysis, cross-validation approach is not suitable for producing an overall confusion matrix. For that reason, we used the Train-test split approach by randomly splitting the dataset into two sets (75% for training – 25% for testing) and then calculated the confusion matrix.

Based on the confusion matrix, there are only 37 misclassified instances for lean class that were predicted as Vulnerable, while the remaining 123 instances were correctly predicted as Clean. For the Vulnerable class, 96 instances were correctly classified, while the remaining 44 instances were misclassified as Clean. Averaged Precision, Recall and F1-score for the Train-test split approach are 0.73, 0.73 and 0.73 respectively, indicating that the predictive performance of algorithm is quite satisfactory.

The results of the present analysis suggest that the construction of relatively precise and accurate class-level vulnerability prediction models based on TD indicators is feasible

Table 15. Train-test split confusion matrix of class-level vulnerability prediction model using the Random Forest algorithm.

	0: Clean	1: Vulnerable
0: Clean	**123**	37
1: Vulnerable	44	**96**

and that Random Forest is the best performing ML algorithm for this purpose. Therefore, this provides us with preliminary evidence for the capacity of TD indicators to indicate the existence of actual vulnerabilities in software classes (i.e. at class-level of granularity).

5. Threats to validity

In the present section we discuss the validity threats of the present study and how our work attempts to mitigate these threats. Emphasis is given on three broader categories of threats to validity, namely (i) External Validity, (ii) Internal Validity, (iii) Construct Validity and (iv) Reliability Validity.

5.1. External validity

External Validity refers to the ability to generalise the results of a given study. The results of the present study are unavoidably subject to external validity threats, due to the fact that the applicability of the selected ML models to predict either the SRL of a software project (i.e. project-level analysis) or the existence of actual vulnerabilities in software classes (i.e. class-level analysis) was examined on specific samples of 210 software projects and 1200 software classes respectively. It is always possible that another set of software projects or classes may exhibit different phenomena and characteristics, which may influence the produced results. However, as far as the project-level analysis is concerned, the selected software projects are quite diverse with respect to their application domain, size, etc., which partially mitigates threats to generalisation. Regarding class-level analysis, the OWASP Benchmark was utilised, which is a well-known vulnerability benchmark used for assessing the accuracy of actual code analysers. Hence, it can be considered a representative dataset of vulnerable and clean software components, containing vulnerabilities that can be found in real-world software applications. In addition, a large part of the proposed methodology consists of constructing prediction models that learn from TD indicators and therefore can be easily adapted to any software application, as long as sufficient and reliable TD-related data are available.

A similar threat stems from the fact that both the dataset used in the project-level analysis and the vulnerability dataset used in the class-level analysis were constructed based on open-source code written in Java programming language. However, the process of building project-level SRL predictors and class-level vulnerability prediction models that are described in this paper builds upon the outputs of tools used for computing TD-related metrics (i.e. indicators), which can act as indicators of the quality attribute of software security. This means that the proposed models can be easily adapted to predict security issues of applications that are coded in a different programming language, as long as there are tools that support the extraction of software-related metrics that can act as TD indicators for the respective language. This also contributes to mitigating threats to generalisation. However, since the dataset does not include code retrieved from industrial applications, we cannot make any speculation on closed-source applications. Commercial systems as well as other object-oriented programming languages can be the subjects of further research.

Finally, another external validity threat, closely related to the previous ones, refers to feature selection, and specifically to whether the observed impacts of the selected

features are experiment specific. Indeed, as in every ML task, the feature selection process highly depends on the dataset on which it is applied. However, we believe that this threat is sufficiently mitigated, for the reasons described in what follows. First of all, as mentioned previously, we selected datasets that are highly representative both for the case of the project-level and for the case of the class-level analysis. Secondly, in both experiments, a large number of statistical tests have been applied for determining the impact of the selected features on the class attribute, and, in turn, for the selection of the final feature sets (e.g. Spearman's rank correlation, Wilcoxon Rank Sum Test, etc.). Hence, feature selection was based on statistical tests and not on heuristics. This provides us with confidence that the selected features (i.e. TD indicators) are closely related to the class attribute (i.e. security risks) in a statistically significant manner, and therefore the observed relationship is not likely to have been caused by chance.

In addition to this, the observed impacts are in-line with the literature. For instance, bugs, i.e. the feature which was found to be the predictor with the highest impact in our analyses, are widely believed to be related to the existence of security issues (Zheng et al. 2006; Austin, Holmgreen, and Williams 2013; Holzmann 2017; Felderer et al. 2016), whereas the existence of quality issues (e.g. code smells, etc.) in security contributing commits (and vice versa) is also highly discussed (Mohammed et al. 2017; Johnson et al. 2013; Holzmann 2017). Recently, the close relationship between TD and security has formally been expressed by several experts in the field (Rindell, Bernsmed, and Jaatun 2019; Rindell and Holvitie 2019; Izurieta et al. 2018; Izurieta and Prouty 2019), whereas software metrics (which are indirect TD indicators), as already discussed in Section 2, have been found to have a close relationship with the existence of security issues in software (Shin and Williams 2008a, 2008b; Chowdhury and Zulkernine 2010; Moshtari and Sami 2016; Siavvas, Kehagias, and Tzovaras 2017; Ferenc et al. 2019; Jimenez et al. 2019; Zhang et al. 2019). Finally, it should be stated that future replications of the present work on different datasets and programming languages, apart from examining the generalisability of the present observations, it would also provide additional feedback with respect to which of these TD indicators are consistently related to security issues, and which not.

5.2. Internal validity

Concerning the *internal validity*, i.e. the possibility of having unwanted or unanticipated relationships between the parameters that might affect the variable that we are trying to predict, it is reasonable to assume that numerous other metrics that affect TD might have not been taken into consideration for constructing both our project-level SRL predictors (see Section 3) and class-level vulnerability prediction models (see Section 4). However, the fact that we constructed our initial set of TD predictors based on software-related metrics that have been widely used in the literature as direct or indirect indicators of TD, such as OO software metrics, code smells and code issues extracted from static analysis tools, limits this thread.

Regarding the final selection of predictors, in the case of project-level prediction, if we had limited our feature selection analysis to only correlations between the security risk levels (SRLs) and TD indicators, then there would have been a threat to internal validity. However, we attempted to mitigate this threat through the complimentary usage of four different feature selection methods to further explore the relationships between the

dependent and independent variables. Respectively, in the case of class-level vulnerability prediction, we have applied two statistical tests with the purpose to examine whether a statistical relationship exists between the studied indicators and the existence of vulnerabilities in software components. More specifically, we initially applied correlation analysis in order to examine whether statistically significant relationships exist between the selected TD indicators and the existence of vulnerabilities in software classes, and, subsequently, discriminant analysis was employed in order to examine whether the selected TD indicators are able to discriminate between vulnerable and clean classes. Based on the results of these statistical tests the final selection of the best features was performed, providing more confidence that internal validity threats are avoided.

5.3. Construct validity

Construct validity refers to the meaningfulness of measurements and that the independent and dependent variables are represented correctly. In this study, the main threats related to construct validity are due to possible inaccuracies in the identification of software-related metrics acting as TD indicators, as well as inaccuracies in the identification of security-related issues and vulnerabilities. However, in order to mitigate the risk, we decided to use a well-known and widely used tool, namely SonarQube, both for the project-level and for the class-level analysis. As already mentioned in Section, SonarQube is widely used in the industry for monitoring software quality, whereas it is the most frequently used tool for measuring TD Principal (Li, Avgeriou, and Liang 2015; Ampatzoglou et al. 2015). In addition to this, it applies a number of security scanners for identifying potential vulnerabilities, while it attempts to reduce the number of the produced false positives. SonarQube is used in the present study as a proof of concept of the proposed methodologies.

The datasets that were used for the purposes of the present study also play an important role in the construct validity of the present work. In order to mitigate the associated risks, for the case of the project-level analysis, a benchmark repository of 210 real-world open-source software applications that were retrieved from GitHub was used as the basis of the experiment. Regarding the class-level analysis, in order to avoid the risks described in Section 4.2 concerning the balance and correctness of the produced vulnerability dataset, the well-accepted OWASP Benchmark test suite was utilised, which is commonly used for the evaluation of static code analysers regarding their ability to detect vulnerabilities. These options are believed to sufficiently mitigate the construct validity threat, as the analysis was based on representative and well-accepted code repositories. As for the experimented prediction models, we exploited the ML algorithms implementation provided by the scikit-learn library, which is widely considered as a reliable tool.

5.4. Reliability validity

Finally, *reliability validity* threats concern the possibility of replicating this study. To facilitate such replication studies, we provide an experimental package containing both the dataset and the scripts that were used for our analysis and prediction model construction. This material can be found online (Online (Last Accessed 29/08/2020)).

6. Implications to researchers and practitioners

Through our study, we have shown that TD indicators that are commonly used for assessing the quality of software products may be sufficient predictors of software security, identified both at project- and class-level of a granularity. More specifically, we have shown that TD indicators can be sufficient predictors of the SRL of software products, whereas they can be also used as the basis for the construction of relatively accurate class-level vulnerability prediction models able to identify software classes that potentially contain actual vulnerabilities. This work has significant implications for both researchers and practitioners, despite the limitations noted in the previous section.

6.1. Implications for researchers

The findings of our present study (both project- and class-level) provide empirical evidence for the capacity of TD to be used as an indicator of software security. This opens a new area of research in the field of TD, for further investigating the potential relationship between TD and Software Security, as well as for finding ways of exploiting the concept of TD for assessing the security of software products. More specifically, researchers could potentially focus on identifying (i) what kind of security implications are imposed by TD, (ii) which TD liabilities are better indicators of underlying security problems, and (iii) how the concepts of TD could be extended in order to be used for assessing software security. Regarding the latter point, some initial attempts for defining the concept of Security Debt have already been made in the literature (Rindell, Bernsmed, and Jaatun 2019; Rindell and Holvitie 2019; Izurieta et al. 2018; Izurieta and Prouty 2019), denoting that there is a potential shift towards extending the concepts of TD into the security realm. It should be noted that the literature in the field of software security lacks a well-accepted methodology for assessing software security (Ansar and Khan 2018; Sentilles, Papatheocharous, and Ciccozzi 2018; Morrison et al. 2018), and TD could be a promising candidate for filling this void.

Apart from the field of TD, we also believe that the findings of the present work impose some implications in the field of vulnerability prediction. The class-level analysis presented in Section 4, and particularly the correlation analysis, discriminant analysis and predictive performance evaluation, revealed that TD indicators (such as code smells, open issues, etc.) can potentially highlight the existence of actual vulnerabilities in software classes. Hence, this opens new research directions in the field of vulnerability prediction. Firstly, researches could further investigate the generalisability of the findings of the present work, by replicating the study using different vulnerability datasets and programming languages. In addition, they could investigate whether the adoption of more advanced ML techniques (e.g. deep learning) could lead to better predictive performance. Finally, another interesting topic would be to investigate whether the performance of existing vulnerability prediction models that are based on other software factors could be improved by enriching these models with TD indicators with observed relationship to the existence of vulnerabilities.

6.2. Implications for practitioners

The production of secure software necessitates the continuous monitoring of the security level of the produced software throughout its overall SDLC and the identification of security issues early enough in the production cycle. Predicting the security level of the software products under development, as well as the potential existence of actual vulnerabilities in their components is critical for the production of secure software, as it enables developers and project managers to make more informed decisions about the overall development. The relevance between TD and Software Security that was empirically observed by the findings of the present work, is very important, as it suggests that quality indicators can potentially be used in order to indicate the existence of security issues that are hidden in the source code of the system and that require individual care.

As far as the project-level analysis is concerned, the results of our work revealed that TD indicators can potentially be used to predict the security level of a software project. In other words, the results of our analysis suggest that TD may also indicate questionable security (i.e. the accumulation of quality issues may indicate the accumulation of security issues). Hence, project managers and developers, by tracking the TD of their software applications under development, are indirectly monitoring the accumulation of potential security risks that reside in software. An approach similar to the one presented in Section 3, would enable the project manager to verify, based on the current TD, what is the security risk level of the application under development compared to real-world applications that are available on the market. This information could be leveraged for making decisions about the actual development. For instance, if the security level of a given project is Low (based on its current TD), the project manager could request an immediate manual security review, in order to detect and fix potential security issues. On the contrary, if the security risk level is high enough, the project manager could postpone a planned security review, and request emphasis to be given on the actual development.

As far as the class-level analysis is concerned, we have shown that TD indicators can potentially discriminate between vulnerable and clean classes, and predict the existence of vulnerabilities in software classes with sufficient level of accuracy. In other words, TD indicators could be used to build prediction models able to highlight security hotspots, i.e. software classes that are likely to contain vulnerabilities. This information is very useful for both the developers and project managers of a software application under development. In fact, this information could be leveraged for better planning their testing and fortification efforts, by allocating limited test resources to high-risk areas (i.e. potentially vulnerable classes). For instance, the testing and refactoring activities could start from those classes that are more likely to contain vulnerabilities. In addition, more exhaustive security testing could be applied to the classes that are marked as vulnerable, in order to increase the possibility of identifying and eventually fixing an underlying vulnerability, and eventually leading to more secure software.

Finally, another benefit that the proposed prediction approaches (both project- and class-level) can offer to practitioners, is that they can be applied from the early stages of software development, and, thus, they can enable the early identification and mitigation of underlying security issues. This can be explained by the fact that they are based on TD indicators, which are computed through static analysis, a software testing mechanism

that (contrary to dynamic analysis) does not require code execution, enabling in that way its early application (Chess and McGraw 2004; Felderer et al. 2016; Mohammed et al. 2017; Do et al. 2017; Nunes et al. 2019). Consequently, the fact that an executable version of the source code is not necessary for the computation of the TD indicators, allows the produced prediction models to be applied very early in the overall development process, even from the first commit. Hence, the application of such models in a frequent manner is expected to help developers detect security risks (e.g. vulnerabilities) in a timely manner and act promptly, ideally prior to the release of the software products, leading to software releases that are bundled with much fewer security issues.

7. Conclusion and future work

In the present paper, we investigate the ability of common TD indicators (e.g. bugs, code smells, etc.) to indicate (i.e. predict) security risks in software products. Emphasis was given both on project-level and on class-level of granularity. For the case of project-level analysis, we examined the ability of TD indicators to predict the security risk level of software projects. For this purpose, a relatively large code repository was constructed, comprising 210 real-world open-source Java applications that were retrieved from GitHub. These applications were then analysed using a popular static analysis platform, i.e. SonarQube, in order to calculate a broad set of TD indicators for each one of the applications of the code repository, as well as their security risk, which was quantified using the Static Analysis Vulnerability Density (SAVD) metric. Subsequently, the SAVD was discretised into Security Risk Levels (SRLs), based on a set of thresholds, which were computed based on real-world data through benchmarking techniques. Several ML models were built having as input the TD indicators and as output the SRLs, in order to evaluate the ability of TD indicators to predict software security risk. Both the cases of binary and 3-class classification were considered, in order to cover different enterprise needs.

In the case of class-level analysis, we examined the ability of TD indicators to predict the existence of actual vulnerabilities in software classes. Initially, we constructed a highly balanced dataset of 1200 vulnerable and clean Java classes retrieved from the OWASP Benchmark, and we analysed these classes with SonarQube in order to compute the studied TD indicators. Subsequently, correlation analysis was employed in order to investigate whether significant relationships exist between the TD indicators and the existence of vulnerabilities, as well as discriminant analysis with the purpose to investigate the ability of the TD indicators to discriminate between vulnerable and clean classes. Finally, several ML models were built, in order to evaluate the capacity of the TD indicators to predict the existence of vulnerabilities in software classes.

Our study led to some interesting observations. In particular, the results of the project-level analysis highlighted the capacity of TD indicators to predict the SRL of software products, whereas Logistic Regression was found to be the best model, demonstrating high predictive performance with an average F1 score greater than 80%. As far as the class-level analysis is concerned, the majority of the selected indicators demonstrated a statistically significant correlation with the existence of vulnerabilities, as well as sufficient power in discriminating between vulnerable and clean software classes. In addition, the produced ML models demonstrated sufficient predictive performance in predicting the existence of vulnerabilities in software classes, with the Random Forest to be the best

performing model, showing an average F1 score of 70.8%. This suggests that TD indicators could potentially be used as the basis for the construction of relatively accurate class-level vulnerability prediction models.

The results of the present study suggest that TD indicators may be sufficient predictors of software security risk both at project-level and at class-level of granularity. Hence, TD may be a sufficient indicator of software security, confirming the findings of our previous empirical study (Siavvas et al. 2019), and providing further support to the recently expressed belief that TD can potentially be used to indirectly measure software security (Rindell, Bernsmed, and Jaatun 2019; Rindell and Holvitie 2019; Izurieta et al. 2018; Izurieta and Prouty 2019).

We believe that the findings of the present study, along with the proposed approaches for TD-based project-level and class-level security risk prediction, are of high importance for software development enterprises. As already mentioned, software development enterprises are seeking for mechanisms able to assist them in identifying and removing potential vulnerabilities early enough in the software development lifecycle, in order to prevent the financial loses and reputation damages that the exploitation of these vulnerabilities may cause to them and to the enterprises that are actually using their software. The present study revealed the relationship between TD and software security, and the ability of TD indicators to also indicate security risks in software, both at project- and at class-level of granularity. Hence, this information could be leveraged by software development enterprises, for making more informed decisions during the actual software development lifecycle, ultimately leading to more secure software.

The project-level analysis suggests that developers and project managers could use TD indicators to get an indication of the overall security risk level of the software product under development. This security risk level, which actually denotes how secure the product is compared to other real-world software products that are available on the market, could be leveraged by project managers to decide promptly whether additional security testing and fortification activities should take place. On the other hand, the class-level analysis suggests that TD indicators could be used to detect security hotspots in a software product, i.e. classes that are likely to contain actual vulnerabilities. This information would allow developers and project managers better plan their testing and fortification activities, by allocating limited test resources (or applying more exhaustive tests) to high-risk areas, hopefully leading to the detection and elimination of actual vulnerabilities. Hence, TD indicators could facilitate the elimination of vulnerabilities and therefore minimise the associated costs that their exploitation may cause.

Accumulated TD is often neglected by project managers, as it is a measure of maintainability, which is often treated as an afterthought in the overall development due to the fact that maintainability issues do not have a visible and immediate impact on the functionality of the produced software. However, throughout the present work we have shown that TD is closely related to software security as it may potentially indicate the existence of security issues (i.e. vulnerabilities) in software. Hence, neglecting TD could potentially lead to the introduction of vulnerabilities and, in turn, to security breaches with devastating consequences for both the software development enterprise and the enterprise that actually uses the compromised software. Hence, we believe that the results of the present study provide additive value to the importance of TD, since linking it with the notion of software security could motivate project managers to change their mindset and hopefully, treat it as an equally important attribute during the development process.

At this point, a statement on the novelty of the present work and specifically on how it manages to advance the state of the art in relevant fields is considered valuable. As far as the overall field of Software Security Assessment is concerned, recently, several researchers have theoretically expressed the relationship between TD and software security and proposed the adoption of TD as the basis for security assessment (Rindell, Bernsmed, and Jaatun 2019; Rindell and Holvitie 2019; Izurieta et al. 2018; Izurieta and Prouty 2019), in an attempt to fill the gap in the field caused by the lack of a well-accepted security measure (Morrison et al. 2018; Ansar and Khan 2018). More specifically, guidelines on how the concept of TD can be extended to support software security have been provided (Rindell, Bernsmed, and Jaatun 2019; Rindell and Holvitie 2019), whereas ways for prioritising security bugs as technical debt items (i.e. quality issues) have already been proposed (Izurieta et al. 2018; Izurieta and Prouty 2019). The present paper extends and complements these research endeavours (which approached the relationship of TD and software security from a theoretical perspective) by providing empirical evidence for the close relationship between these two factors, both at project- and class-level of granularity. The present work also extends them by showcasing the potential feasibility of using the TD Indicators as security risk indicators in practice (and not just as theoretical constructs), specifically through the adoption of ML, opening in that way directions for future experimentation.

Regarding the vulnerability prediction field, as already mentioned, this is the first study that focuses on the ability of TD indicators (like code smells, bugs, etc.) to indicate the existence of vulnerabilities and security risks in general. Recent attempts have shown that other factors, such as text mining (Dam et al. 2018) and software metrics (Yang, Ryu, and Baik 2016; Zhang et al. 2019), as well as their combination (Zhang et al. 2015; Sultana 2017; Jimenez et al. 2019), can lead to promising vulnerability prediction models. Hence, the results of the present study complement previous works, and open new directions for future experimentation, towards investigating whether the incorporation of TD indicators could potentially further improve the accuracy of existing models.

Several directions for future work can be identified. First of all, the present study was based on open-source software applications written in Java programming language. In order to investigate the generalisability of our results, we are planning to replicate the present work by considering software applications written in programming languages other than Java, whereas the case of commercial software applications will be also considered. In addition, in the present study, the SAVD metric was used as a measure of software security risk and the SonarQube static analysis platform was used for its quantification. In the future, we are planning to redo the present analysis using other open-source or commercial static code analysers for quantifying SAVD, while we are also planning to consider other software security risk indicators like the Attack Surface (Howard 2007; Manadhata and Wing 2011). Finally, if the results of the present study are generalised, we are planning to implement our models in the form of individual tools (or as part of common IDEs or software quality platforms), which will facilitate decision making during the overall SDLC, by helping developers and project managers identify and mitigate security risks early enough in the development process.

Notes

1. https://nvd.nist.gov/vuln/detail/CVE-2017-5638

2. https://investor.equifax.com/news-and-events/news/2019/05-10-2019-113504540
3. https://github.com
4. https://owasp.org/www-project-benchmark/
5. https://www.sonarqube.org/
6. https://sonarcloud.io/explore/projects
7. http://findbugs.sourceforge.net/
8. https://pmd.github.io/
9. https://docs.sonarqube.org/latest/user-guide/security-rules/
10. https://scikit-learn.org/stable/index.html
11. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
12. https://owasp.org/www-project-benchmark/
13. As stated in Section 4.2, the TD indicator named *classes* was not considered in the present analysis, since it was not applicable in the context of the class-level analysis.

Acknowledgments

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

Disclosure statement

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

Data availability and supporting material

The data that support the findings of the present work along with additional supporting material of the present study are available at (Online (Last Accessed 29/08/2020)).

Funding

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

ORCID

Miltiadis Siavvas http://orcid.org/0000-0002-3251-8723
Dimitrios Tsoukalas http://orcid.org/0000-0001-9986-0796
Dimitrios Tzovaras http://orcid.org/0000-0001-6915-6722

References

Alhazmi, O. H., Y. K. Malaiya, and I. Ray. 2007. "Measuring, Analyzing and Predicting Security Vulnerabilities in Software Systems." *Computers and Security* 26 (3): 219–228. doi:10.1016/j.cose.2006.10.002.

Alves, N. S. R., T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman. 2016. "Identification and Management of Technical Debt: A Systematic Mapping Study." *Information and Software Technology* 70: 100–121. doi:10.1016/j.infsof.2015.10.008.

Ampatzoglou, A., A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou. 2015. "The Financial Aspect of Managing Technical Debt: A Systematic Literature Review." *Information and Software Technology* 64: 52–73. doi:10.1016/j.infsof.2015.04.001.

Ansar, S. A. A., and R. A. Khan. 2018. "A Phase-wise Review of Software Security Metrics." *Networking Communication and Data Knowledge Engineering*.

Austin, A., C. Holmgreen, and L. Williams. 2013. "A Comparison of the Efficiency and Effectiveness of Vulnerability Discovery Techniques." *Information and Software Technology* 55 (7): 1279–1288. doi:10.1016/j.infsof.2012.11.007.

Bellman, R. E. 2003. "Dynamic Programming." *Dover Books on Computer Science Series. Dover Publications*.

Bishop, C. M. 2006. *Pattern Recognition and Machine Learning*. Springer-Verlag, New York: springer.

Boland, T., and P. E. Black. 2012. "Juliet 1.1 C/C++ and Java Test Suite." *Computer (Long. Beach. Calif.)* 45 (10): 88–90.

Bruntink, M., and A. van Deursen. 2006. "An Empirical Study into Class Testability." *Journal of Systems and Software* 79 (9): 1219–1232. doi:10.1016/j.jss.2006.02.036.

Camilo, F., A. Meneely, and M. Nagappan. 2015. "Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project." In 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 269–279. Florence, Italy: IEEE.

Chawla, N. V., K. W. Bowyer, L. O. Hall, and W. Philip Kegelmeyer. 2002. "SMOTE: Synthetic Minority Over-sampling Technique." *Journal of Artificial Intelligence Research* 16: 321–357. doi:10.1613/jair.953.

Chess, B., and G. McGraw. 2004. "Static Analysis for Security." *Security & Privacy, IEEE* 2: 76–79. doi:10.1109/MSP.2004.111.

Chidamber, S. R., and C. F. Kemerer. 1994. "A Metrics Suite for Object Oriented Design." *IEEE Transactions on Software Engineering* 20 (6): 476–493. doi:10.1109/32.295895.

Chowdhury, I., and M. Zulkernine. 2010. "Can Complexity, Coupling, and Cohesion Metrics Be Used as Early Indicators of Vulnerabilities?" In Proceedings of the 2004 ACM symposium on Applied computing, Sierre, Switzerland.

Chowdhury, I., and M. Zulkernine. 2011. "Using Complexity, Coupling, and Cohesion Metrics as Early Indicators of Vulnerabilities." *Journal of Systems Architecture* 57: 294–313. doi:10.1016/j.sysarc.2010.06.003.

Cohen, J. 2013. *Statistical Power Analysis for the Behavioral Sciences*. New York, USA: Academic press.

Cunningham, W. 1993. "The WyCash Portfolio Management System." *ACM SIGPLAN OOPS Messenger* 4 (2): 29–30. doi:10.1145/157710.157715.

Dam, H. K., T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose. 2018. "Automatic Feature Learning for Predicting Vulnerable Software Components." *IEEE Transactions on Software Engineering*, (pp. 1–1).

Digkas, G., M. Lungu, A. Chatzigeorgiou, and P. Avgeriou. 2017. "The Evolution of Technical Debt in the Apache Ecosystem." In European Conference on Software Architecture (ECSA), 51–66. Canterbury, UK: Springer.

Do, L., N. Quang, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill. 2017. "Just-in-time Static Analysis." In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 307–317. Santa Barbara, CA: ACM.

Elish, M. O., and K. O. Elish. 2009. "Application of TreeNet in Predicting Object-Oriented Software Maintainability: A Comparative Study." In 2009 13th European Conference on Software Maintenance and Reengineering (CSMR), 69–78, March. ISSN: 1534-5351.

Eski, S., and F. Buzluca. 2011. "An Empirical Study on Object-Oriented Metrics and Software Evolution in order to Reduce Testing Costs by Predicting Change-Prone Classes." In 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), 566–571, March. ISSN: null.

Felderer, M., M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner. 2016. "Security Testing: A Survey." In *Advances in Computers*, edited by Atif Memon, Vol. 101, 1–51. Elsevier.

Ferenc, R., P. Hegedüs, P. Gyimesi, G. Antal, D. Bán, and T. Gyimóthy. 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, Montreal, Quebec, Canada.

Feurer, M., A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. 2015. "Efficient and Robust Automated Machine Learning." *Advances in Neural Information Processing Systems*, 2962–2970.

Fioravanti, F., and P. Nesi. 2001. "Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-oriented Systems." *IEEE Transactions on Software Engineering* 27 (12): 1062–1084.

Fowler, M. 2018. *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Professional.

Gegick, M., L. Williams, J. Osborne, and M. Vouk. 2008. "Prioritizing Software Security Fortification through Code-Level Metrics." Proceedings of the 4th ACM workshop on Quality of Protection, Alexandria, Virginia, USA, 31–38.

Giger, E., M. Pinzger, and H. C. Gall. 2012. "Can We Predict Types of Code Changes? An Empirical Analysis." In 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), 217–226, June. ISSN: 2160-1852.

Griffith, I., D. Reimanis, C. Izurieta, Z. Codabux, A. Deo, and B. Williams. 2014. "The Correspondence between Software Quality Models and Technical Debt Estimation Approaches." In *Sixth International Workshop on Managing Technical Debt (MTD)*, 19–26. Victoria, BC, Canada: IEEE.

Heitlager, I., T. Kuipers, and J. Visser. 2007. "A Practical Model for Measuring Maintainability." 6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007), Lisbon, Portugal, 30–39.

Holzmann, G. J. 2017. "The Value of Doubt." *IEEE Software* 34 (1): 106–109. doi:10.1109/MS.2017.19.

Howard, M. 2007. "Determining Relative Attack Surface." US Patent 7,299,497, November 20.

Howard, M., D. LeBlanc, and J. Viega. 2010. *24 Deadly Sins of Software Security*. NY, United States: McGraw-Hill.

Izurieta, C., and M. Prouty. 2019. "Leveraging Secdevops to Tackle the Technical Debt Associated with Cybersecurity Attack Tactics." In Proceedings of the 2nd International Conference on Technical Debt, Montreal, Quebec, Canada.

Izurieta, C., D. Rice, K. Kimball, and T. Valentien. 2018. "A Position Study to Investigate Technical Debt Associated with Security Weaknesses." In 2018 International Conference on Technical Debt, Gothenburg, Sweden.

Jimenez, M., R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman. 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, Tallinn, Estonia.

Johnson, B., Y. Song, E. Murphy-Hill, and R. Bowdidge. 2013. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In 2013 35th International Conference on Software Engineering (ICSE), 672–681. San Francisco, CA: IEEE.

Kirasich, K., T. Smith, and B. Sadler. 2018. "Random Forest Vs Logistic Regression: Binary Classification for Heterogeneous Datasets."

Kosti, M. V., A. Ampatzoglou, A. Chatzigeorgiou, G. Pallas, I. Stamelos, and L. Angelis. 2017. "Technical Debt Principal Assessment Through Structural Metrics." In 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Vienna, Austria, 329–333.

Krsul, I. 1998. "Software Vulnerability Analysis." PhD diss., Department of Computer Sciences, Purdue University.

Letouzey, J.-L., and M. Ilkiewicz. 2012. "Managing Technical Debt with the Sqale Method." *IEEE Software* 29 (6): 44–51. doi:10.1109/MS.2012.129.

Li, W., and S. Henry. 1993. "Object-oriented Metrics that Predict Maintainability." *Journal of Systems and Software* 23 (2): 111–122. doi:10.1016/0164-1212(93)90077-B.

Li, Z., P. Avgeriou, and P. Liang. 2015. "A Systematic Mapping Study on Technical Debt and Its Management." *Journal of Systems and Software* 101: 193–220. doi:10.1016/j.jss.2014.12.027.

Luszcz, J. 2018. "Apache Struts 2: How Technical and Development Gaps Caused the Equifax Breach." *Network Security* 2018 (1): 5–8. doi:10.1016/S1353-4858(18)30005-9.

Manadhata, P. K., and J. M. Wing. 2011. "An Attack Surface Metric." *IEEE Transactions on Software Engineering* 37 (3): 371–386. doi:10.1109/TSE.2010.60.

Marinescu, R. 2012. "Assessing Technical Debt by Identifying Design Flaws in Software Systems." *IBM Journal of Research and Development* 56 (5): 9. doi:10.1147/JRD.2012.2204512.

McGraw, G. 2006. "Software Security: Building Security In." *Addison-Wesley Prof*.

Mohammed, N. M., M. Niazi, M. Alshayeb, and S. Mahmood. 2017. "Exploring Software Security Approaches in Software Development Lifecycle: A Systematic Mapping Study." *Computer Standards & Interfaces* 50: 107–115. doi:10.1016/j.csi.2016.10.001.

Morrison, P., K. Herzig, B. Murphy, and L. Williams. 2015. "Challenges with Applying Vulnerability Prediction Models." In Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, Urbana, Illinois, 1–9.

Morrison, P., D. Moye, R. Pandita, and L. Williams. 2018. "Mapping the Field of Software Life Cycle Security Metrics." *Information and Software Technology* 102 (May): 146–159. doi:10.1016/j.infsof.2018.05.011.

Moshtari, S., and A. Sami. 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, Pisa, Italy, 1415–1421.

Moshtari, S., A. Sami, and M. Azimi. 2013. "Using Complexity Metrics to Improve Software Security." *Computer Fraud & Security* 2013 (5): 8–17. doi:10.1016/S1361-3723(13)70045-9.

Mosteller, F., and J. W. Tukey. 1968. "Data Analysis, Including Statistics." *Handbook of Social Psychology* 2: 80–203.

Munaiah, N., and A. Meneely. 2016. "Beyond the Attack Surface: Assessing Security Risk with Random Walks on Call Graphs." Proceedings of the 2016 ACM Workshop on Software PROtection, 3–14. doi:10.1145/2995306.2995311.

Neuhaus, S., T. Zimmermann, C. Holler, and A. Zeller. 2007. "Predicting Vulnerable Software Components." Proceedings of the 14th ACM conference on Computer and communications security CCS 07, Alexandria, Virginia, USA, 529.

Nugroho, A., J. Visser, and T. Kuipers. 2011. "An Empirical Model of Technical Debt and Interest." In Proceedings of the 2nd Workshop on Managing Technical Debt, 1–8. Waikiki, Honolulu HI: ACM.

Nunes, P., I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira. 2019. "An Empirical Study on Combining Diverse Static Analysis Tools for Web Security Vulnerabilities Based on Development Scenarios." *Computing* 101 (2): 161–185. doi:10.1007/s00607-018-0664-z.

Online. (Last Accessed 29/08/2020). "Supporting Material." https://sites.google.com/view/technical-debt-as-an-indicator/main

Palomba, F., G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. 2018. "On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation." *Empirical Software Engineering* 23 (3): 1188–1221. doi:10.1007/s10664-017-9535-z.

Pang, Y., X. Xue, and H. Wang. 2017. "Predicting Vulnerable Software Components through Deep Neural Network." Proceedings of the 2017 International Conference on Deep Learning Technologies - ICDLT '17, Chengdu, China, 6–10.

Riaz, M., E. Mendes, and E. Tempero. 2009. "A Systematic Review of Software Maintainability Prediction and Metrics." In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, 367–377. Lake Buena Vista, FL: IEEE Computer Society.

Rindell, K., K. Bernsmed, and M. G. Jaatun. 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, Canterbury CA, United Kingdom.

Rindell, K., and J. Holvitie. 2019. "Security Risk Assessment and Management as Technical Debt." In International Workshop on Secure Software Engineering in DevOps and Agile Development, Oxford, UK.

Roumani, Y., J. K. Nwankpa, and Y. F. Roumani. 2016. "Examining the Relationship between Firm's Financial Records and Security Vulnerabilities." *International Journal of Information Management* 36: 987–994. doi:10.1016/j.ijinfomgt.2016.05.016.

Scandariato, R., J. Walden, A. Hovsepyan, and W. Joosen. 2014. "Predicting Vulnerable Software Components via Text Mining." *IEEE Transactions on Software Engineering* 40 (10): 993–1006. doi:10.1109/TSE.2014.2340398.

Sentilles, S., E. Papatheocharous, and F. Ciccozzi. 2018. "What Do We Know about Software Security Evaluation? A Preliminary Study." In 6th International Workshop on Quantitative Approaches to Software Quality, Nara, Japan.

Shatnawi, R., and L. Wei. 2008. "The Effectiveness of Software Metrics in Identifying Error-prone Classes in Post-release Software Evolution Process." *Journal of Systems and Software* 81 (11): 1868–1882. doi:10.1016/j.jss.2007.12.794.

Shin, Y., A. Meneely, L. Williams, and J. A. Osborne. 2011. "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities." *IEEE Transactions on Software Engineering* 37 (6): 772–787. doi:10.1109/TSE.2010.81.

Shin, Y., and L. Williams. 2008a. "Is Complexity Really the Enemy of Software Security.", Proc. the 4th ACM Workshop on Quality of Protection. Alexandria, Virginia, USA, October.

Shin, Y., and L. A. Williams. 2008b. "An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics." In ESEM'08: Proceedings of the 2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Kaiserslautern, Germany.

Shin, Y., and L. Williams. 2013. "Can Traditional Fault Prediction Models Be Used for Vulnerability Prediction?" *Empirical Software Engineering* 18 (1): 25–59. doi:10.1007/s10664-011-9190-8.

Siavvas, M., K. Chatzidimitriou, and A. Symeonidis. 2017. "QATCH - an Adaptive Framework for Software Product Quality Assessment." *Expert Systems with Applications* 86: 350–366. doi:10.1016/j.eswa.2017.05.060.

Siavvas, M., E. Gelenbe, D. Kehagias, and D. Tzovaras. 2018a. "Static Analysis-Based Approaches for Secure Software Development." In *Security in Computer and Information Sciences*, 142–157. Cham: Springer International Publishing.

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

Siavvas, M., D. Kehagias, and D. Tzovaras. 2017. "A Preliminary Study on the Relationship among Software Metrics and Specific Vulnerability Types." International Conference on Computational Science and Computational Intelligence (CSCI) 2017, Las Vegas, Nevada, USA, 916–921.

Siavvas, M., D. Tsoukalas, M. Jankovic, D. Kehagias, A. Chatzigeorgiou, D. Tzovaras, N. Anicic, and E. Gelenbe. 2019. "An Empirical Evaluation of the Relationship between Technical Debt and Software Security." 9th International Conference on Information Society and Technology (ICIST) 2019, Kopaonik, Serbia.

Siebra, C. A., A. Cavalcanti, F. Q. B. Silva, A. L. M. Santos, and T. B. Gouveia. 2014. "Applying Metrics to Identify and Monitor Technical Debt Items during Software Evolution." In 2014 IEEE International Symposium on Software Reliability Engineering Workshops, Naples, Italy, 92–95.

Singh, Y., and A. Saha. 2012. "Prediction of Testability Using the Design Metrics for Object-Oriented Software." *International Journal of Computer Applications in Technology* 44 (1): 12–22. Geneva 15, CHE Publisher: Inderscience Publishers. doi:10.1504/IJCAT.2012.048204.

Sultana, K. Z. 2017. "Towards a Software Vulnerability Prediction Model Using Traceable Code Patterns and Software Metrics." In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana-Champaign, IL, USA, 1022–1025.

Tang, Y., F. Zhao, Y. Yang, H. Lu, Y. Zhou, and B. Xu 2015. "Predicting Vulnerable Components via Text Mining or Software Metrics? An Effort-aware Perspective." In 2015 IEEE International Conference on Software Quality, Reliability and Security, 27–36. Vancouver, Canada: IEEE.

Tsoukalas, D., D. Kehagias, M. Siavvas, and A. Chatzigeorgiou. 2020. "Technical Debt Forecasting: An Empirical Study on Open-source Repositories." *Journal of Systems and Software* 170: 110777. doi:10.1016/j.jss.2020.110777.

Tsoukalas, D., M. Siavvas, M. Jankovic, D. Kehagias, A. Chatzigeorgiou, and D. Tzovaras. 2018. "Methods and Tools for TD Estimation and Forecasting: A State-of-the-art Survey." In International Conference on Intelligent Systems (IS 2018). Madeira, Portugal: IEEE.

Van Koten, C., and A. R. Gray. 2006. "An Application of Bayesian Network for Predicting Object-oriented Software Maintainability." *Information and Software Technology* 48 (1): 59–67. doi:10.1016/j.infsof.2005.03.002.

Wagner, S., K. Lochmann, L. Heinemann, M. Klas, A. Trendowicz, R. Plosch, A. Seidi, A. Goeb, and J. Streit. 2012. "The Quamoco Product Quality Modelling and Assessment Approach." 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 1133–1142.

Walden, J., and M. Doyle. 2012. "SAVI: Static-Analysis Vulnerability Indicator." *IEEE Security & Privacy* 10 (3): 32–39. doi:10.1109/MSP.2012.1.

Walden, J., M. Doyle, G. A. Welch, and M. Whelan. 2009. "Security of Open Source Web Applications." 2009 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, FL.

Walden, J., J. Stuckman, and R. Scandariato. 2014. "Predicting Vulnerable Components: Software Metrics Vs Text Mining." Proceedings - International Symposium on Software Reliability Engineering, Naples, Italy.

Xuan, J., Y. Hu, and H. Jiang. 2017. "Debt-Prone Bugs: Technical Debt in Software Maintenance." *Computing Research Repository (Corr)* abs/1704.04766. http://arxiv.org/abs/1704.04766

Yang, J., D. Ryu, and J. Baik. 2016. "Improving Vulnerability Prediction Accuracy with Secure Coding Standard Violation Measures." 2016 International Conference on Big Data and Smart Computing, BigComp 2016, Hong Kong, China, 115–122.

Zazworka, N., R. O. Spínola, A. Vetro, F. Shull, and C. Seaman. 2013. "A Case Study on Effectively Identifying Technical Debt." In Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering, 42–47. Porto de Galinhas, Brazil: ACM.

Zhang, M., X. C. de Carné, L. Wang, and A. Ragab. 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. doi:10.1109/TIFS.2019.2895963.

Zhang, Y., D. Lo, X. Xia, B. Xu, J. Sun, and S. Li 2015. "Combining Software Metrics and Text Features for Vulnerable File Prediction." In 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS), 40–49. Gold Coast, Australia: IEEE.

Zheng, J., L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. 2006. "On the Value of Static Analysis for Fault Detection in Software." *IEEE Transactions on Software Engineering* 32 (4): 240–253. doi:10.1109/TSE.2006.38.

Zhou, Y., and H. Leung. 2007. "Predicting Object-oriented Software Maintainability Using Multivariate Adaptive Regression Splines." *Journal of Systems and Software* 80 (8): 1349–1361. doi:10.1016/j.jss.2006.10.049.

Zhou, Y., H. K. N. Leung, Q. Song, J. Zhao, H. Lu, L. Chen, and B. Xu. 2012. "An In-depth Investigation into the Relationships between Structural Metrics and Unit Testability in Object-oriented Systems." *Science China Information Sciences* 55: 2800–2815. doi:10.1007/s11432-012-4745-x.

Zhou, Y., and B. Xu. 2008. "Predicting the Maintainability of Open Source Software Using Design Metrics." *Wuhan University Journal of Natural Sciences* 13 (1): 14–20. doi:10.1007/s11859-008-0104-6.