The Journal of Systems & Software 198 (2023) 111621

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

# Trustworthiness models to categorize and prioritize code for security improvement☆

Nadia Medeiros [a,*], Naghmeh Ivaki [a], Pedro Costa [a,b], Marco Vieira [a]

[a] CISUC, Department of Informatics Engineering, University of Coimbra, Portugal
[b] ISCAC, Polytechnic Institute of Coimbra, Portugal

ARTICLE INFO

*Article history:*
Received 5 August 2022
Received in revised form 14 December 2022
Accepted 16 January 2023
Available online 18 January 2023

*Keywords:*
Software security
Code security review
Code categorization
Trustworthiness model
Software metrics
Machine learning

ABSTRACT

The exploitation of software security vulnerabilities can have severe consequences. Thus, it is crucial to devise new processes, techniques, and tools to support teams in the development of secure code from the early stages of the software development process, while potentially reducing costs and shortening the time to market. In this paper, we propose an approach that uses security evidences (e.g., software metrics, bad smells) to feed a set of trustworthiness models, which allow characterizing code from a security perspective. In practice, the goal is to identify the code units that are more prone to be vulnerable (i.e., are less trustworthy from a security perspective), thus helping developers to improve their code. A clustering-based approach is used to categorize the code units based on the combination of the scores provided by several trustworthiness models and taking into account the criticality of the code. To instantiate our proposal, we use a dataset of software metrics (e.g., CountLine, Cyclomatic Complexity, Coupling Between Objects) for files and functions of the Linux Kernel and Mozilla Firefox projects, and a set of machine learning algorithms (i.e., Random Forest, Decision Tree, SVM Linear, SVM Radial, and Xboost) to build the trustworthiness models. Results show that code that is more prone to be vulnerable can be effectively distinguished, thus demonstrating the applicability and usefulness of the proposed approach in diverse scenarios.



## 1. Introduction

Nowadays, almost every daily activity, from entertaining games to the control systems of nuclear sites, is backed by complex software, prone to be vulnerable; thus, a successful security attack has consequences more severe than ever before. Beyond the harmful and non-negligible impact on reputation, attacks frequently lead to catastrophic failures in the target systems, sensitive data breaches, financial losses, and even safety violations. This jeopardizes the trust of end-users, customers, organizations, and businesses in software systems.

Despite the considerable effort and investment in building secure and defensive systems, the number of breaches in software and the number of attacks is increasing tremendously. Since the beginning of the COVID-19 pandemic, the FBI reported a 300% increase in cybercrimes.[1] This increase in security attacks has led to a considerable gap between the ability of attackers and the available security skills of software developers, making security a day-to-day struggle for every software development team.

Research studies show that most software vulnerabilities are either caused by the use of legacy code or by software configuration, design, and implementation mistakes made by less professional or negligent developers with a lack of knowledge about security (Graff and Wyk, 2003). Existing techniques and tools to discover vulnerabilities or bugs in software are known to be inaccurate, reporting nonexistent vulnerabilities or missing the existing ones (Araujo Neto and Vieira, 2013), forcing intensive and global code review/correction time-consuming tasks, with the consequent increase of costs, and therefore proving to be ineffective and of little use. This way, it becomes of utmost importance to avoid or eliminate software vulnerabilities in the early phases of the software development process, as the later the faults are discovered in the lifecycle, the greater are the consequences and the costs of fixing them.

To improve the current situation, we need to identify, investigate, and use the early evidences of security issues in the code, in a way that supports developers in the detection of potential issues during the software development process (i.e., design and implementation) (Evans and Larochelle, 2002). This paper proposes an **approach based on the use of *software security***

☆ Editor: Alexander Chatzigeorgiou.
* Corresponding author.
*E-mail addresses:* nadiam@dei.uc.pt (N. Medeiros), naghmeh@dei.uc.pt (N. Ivaki), pncosta@dei.uc.pt (P. Costa), mvieira@dei.uc.pt (M. Vieira).
[1] 2020 Internet Crime Report, reported by the FBI's Internet Crime Complaint Center, available on https://www.ic3.gov/Media/PDF/AnnualReport/2020_IC3Report.pdf.

https://doi.org/10.1016/j.jss.2023.111621
0164-1212/

***evidences*, not for predicting or detecting vulnerabilities, but for *assessing the trustworthiness level (or category)* of code units (e.g., files/classes or functions/methods)** and thus call the attention of developers to the most untrustworthy (potentially insecure or vulnerable) ones. The proposed approach uses the clustering technique and suggests categorizing the code units into several categories (instead of only two: vulnerable and non-vulnerable) to help the developers prioritize the code units to be reviewed based on the available resources. To instantiate our approach, we focus on software metrics as security evidences as these can be collected at any time during the software development process and are commonly used as indicators of software quality (Rawat et al., 2012).

We **build code classification (categorization) models, not directly based on the values of software metrics, but on the trustworthiness scores computed by several trustworthiness models (e.g., machine learning algorithms in this study) that are created (or trained) over several software metrics.** In practice, our solution does not identify vulnerable code but can be used instead to warn developers about the units of code that seem to be more untrustworthy. By assigning a trustworthiness level to each unit of code, it is then up to the developers to decide which levels (categories) need more attention, depending on the criticality of the system being developed and on the available resources (e.g., human resources, financial resources, time), which makes our proposal suitable for different application scenarios. Trustworthiness level refers to the extent to which a piece of software can be trusted. The trustworthiness of a code unit can be determined by the combination of pieces of evidence of software security, showing that it is trustworthy (Araujo Neto and Vieira, 2013). Since security evidences are used to determine the trustworthiness level, an untrustworthy piece of code can be considered as code that is more likely to be vulnerable.

To instantiate the proposed approach, we used a dataset built by Alves et al. (2016), containing a large number of software metrics of different types (e.g., complexity, volume, coupling, and cohesion) for different code units (files and functions) of the Linux Kernel and Mozilla Firefox projects, and the security vulnerabilities reported for those projects from 2000 to 2016. Several machine learning algorithms (Random Forest (RF), Decision Tree (DT), Linear and Radial Support Vector Machine (SVM), and Extreme Gradient Boosted (EGB)) were used to build the trustworthiness models. Each model assigns one score to each code unit, and code units are categorized by applying a clustering technique, the **Clustering Large Application (CLARA)** (Gupta and Panda, 2019), on the outputs of the different trustworthiness models.

We performed two sets of experiments to attest to the proposed approach and the results obtained. First, we tested the approach in an unknown dataset (different from the one used for training and building the trustworthiness models) to study whether the same pattern is observed in the results. To do so, we collected the code units of both Linux kernel and Mozilla Firefox projects with vulnerabilities reported after 2017 (the training dataset only includes vulnerabilities reported until 2016). Second, we validated the results by using an expert-based approach. The experts, provided with the source code of several selected vulnerable and non-vulnerable files and functions and the available information regarding the software metrics, were asked to rank the files and functions based on their perceived trustworthiness. The obtained trustworthiness level from the experts was then compared with the category of the code obtained from the clustering.

In practice, our experimental evaluation intends to contribute to answering the following Research Questions (RQs):

- **RQ1**. Is it possible (and how) to integrate diverse trustworthiness models, from a security perspective, to categorize code into different categories with different trustworthiness levels?
- **RQ2**. Can the proposed approach (and how) be applied in different software application scenarios with different security concerns?

The results show that the categories identified as less trustworthy have, in fact, a higher percentage of files/functions reported as vulnerable (as validated using the unknown dataset). In addition, by comparing the results with the expert-based ranking, we observed that the code units categorized as less trustworthy are also considered untrustworthy by the experts (answering to RQ1). Moreover, the results show that adjusting the number of clusters is important when different application scenarios are considered (answering to RQ2). In summary, our approach can be used to categorize software code units considering their level of trustworthiness (identified by software metrics combined with machine learning-based models). This allows identifying untrustworthy code and warn developers about the code units that seem to be more prone to be vulnerable. The proposed model can be adapted when different level of resources to review code and detect vulnerabilities are available. The main contributions of this work can be summarized as follows:

- An approach to categorize code units in categories with different trustworthiness levels, from a security perspective. The approach is generic and can be applied to diverse software security evidences and diverse trustworthiness models to score code units. To the best of our knowledge, there is no software trustworthiness model in the literature that is able to score and characterize code units from a security perspective during the software development phase, aiming to help developers improving the code.
- Instantiate of the proposed approach using software metrics as evidences and machine learning-based prediction models as trustworthiness model.
- The practical application of the proposed approach over a dataset that includes the source code of two well-known representative projects (Linux Kernel and Mozilla Firefox) and the use of a large number of software metrics of different types (e.g., complexity, volume, coupling, and cohesion) for different code units (files and functions) and the security vulnerabilities reported for those projects

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 presents the proposed approach and Section 4 describes experimental evaluation. Results are presented in Section 5 and Section 6 discusses the threats to validity. Section 7 concludes the paper and presents ideas for future work.

## 2. Related work

This section reviews related works on software trustworthiness assessment, software security assessment, vulnerability detection, and the use of software metrics for vulnerability detection and software trustworthiness assessment.

### 2.1. Software trustworthiness assessment

Software trustworthiness is an important concern for developers, researchers, and enterprises. However, several factors make assessing trustworthiness a nontrivial task. These factors include the diversity of software systems, the large scale and high complexity of today's systems, and the subjective notion of trust and

trustworthiness, because, depending on the context (e.g., critical systems or non-critical systems), different quality attributes (e.g., security or performance) may be involved in the assessment of the trustworthiness level of a system (Medeiros et al., 2018). The complex nature of trust in computer systems triggered many research work, as discussed next.

A survey is conducted in Del Bianco et al. (2011) to understand the factors that influence trust in open source software (OSS) by users and developers. A total of 151 OSS stakeholders with different roles and responsibilities participated in the survey. The survey results show that functionality and reliability are the most critical factors.

In a different kind of study (Alarcon et al., 2017a), transparency and reputation were studied as factors that may influence trust perceptions as well as time spent reviewing code by professional software developers. In a previous study, the same authors (Alarcon et al., 2017b) explored how developers assess code trustworthiness when asked to reuse an existing code base. They used an expert-based analysis to explore experienced programmers' perspectives on code reuse, and concluded that implementing software when considering factors like reputation, transparency, and performance can influence the trust in reusing existing code. In their second study (Alarcon et al., 2017a), their findings suggest that the influence of transparency on trust perceptions are not as strong and straightforward as previously thought.

A trustworthiness model based on Structural equation modeling (SEM) is proposed in Deng et al. (2019). They consider several attributes, including functionality, reliability, safety, maintainability, and several sub-attributes for each. Then, they use SEM to obtain their weights by performing a survey.

Despite the merit of these works, they are mainly focused on expert-based analysis to assess the trustworthiness of the code. Although this helps to understand the essential factors influencing trustworthiness, it cannot be applied in an automatic way and on a bigger scale. The other shortcoming of these works compared to what we are presenting in this paper is that they do not consider security as one of the main factors of software trustworthiness.

In addition to the above works, trust and trustworthiness assessment are vastly explored in the context of complex and dynamic environments, such as Cloud (Medeiros et al., 2017b; Horvath and Agrawal, 2015; Lee and Brink, 2020). However, most of the works in this context are customer-centric and do not address the improvement of the software under development, especially from a security perspective.

### 2.2. Software security assessment

Several security evaluation methods have been proposed over the years in order to enhance the security of software systems. For example, the Orange Book (Qiu et al., 1985) and the Common Criteria for Information Technology Security Evaluation (SCSUG, 2001) define a set of generic rules that allow developers to specify the security attributes of their products and evaluators to verify if products actually meet their requirements. Therefore, a lot of efforts have been made by researchers trying to discover attributes that can characterize the overall security of a system. These attributes can represent security flaws, i.e. defects in a software component that, when combined with the necessary conditions, can lead to a software vulnerability. Of particular interest are attributes that are associated with known exploits and known mitigation techniques (Seacord and Householder, 2005). However, representative security metrics are hard to obtain as they involve making isolated estimations about the ability of a hacker to exploit unknown vulnerabilities (Neto and Vieira, 2011).

Developers are expected to put the necessary effort to apply adequate software security principles and rules during the development of software systems in order to minimize the probability of the existence of software vulnerabilities (that can be exploited by security attacks). Therefore, we can find a lot of related work in the literature regarding the definition of best practices, standards, and regulations to help developers in building secure software (e.g., ISO/IEC 27000 Disterer, 2013a, ISO 15408 Potii et al., 2015, Software Quality Assurance (SQA) Galin, 2004; Chemuturi, 2010; Heimann, 2014, OWASP secure coding practices Turpin, 2010; Marcil, 2014, ISO/IEC 27034 Poulin and Guay, 2008, and Privacy by Design (PbD) Cavoukian, 2009) (Jung et al., 2004; I. ISO and I. Std, 2009; Disterer, 2013b). A comparison between most of these efforts can be found in Beckers et al. (2014) and Shan et al. (2019).

### 2.3. Vulnerability detection

We can find several studies in the literature that are focused on the detection and elimination of vulnerabilities during the software development process (Evans and Larochelle, 2002; Graff and Wyk, 2003). The well-known techniques, such as, static code analysis (Chess and McGraw, 2004) and penetration testing (Arkin et al., 2005) are frequently used in these works. In static code analysis, there is no need to execute the code, and the source code is examined statically (Chess and McGraw, 2004), either manually or by using static analysis tools (SATs). Manual examination of code requires skilled code auditors with profound knowledge regarding security vulnerabilities and security attacks, and the process of examination is time-consuming. In contrast, SATs do not require highly skilled human auditors with security expertise as they encapsulate security knowledge into the tool; thus, they are faster. Nevertheless, the output of these tools still should be analyzed and confirmed by experts as they produce a large number of false alarms.

In contrasts to code static analysis, penetration testing can be applied when the code can already be executed (Arkin et al., 2005). Its objective is to check for exploitable vulnerabilities through emulating security attacks. Despite the usefulness of this technique, many vulnerabilities remain undisclosed until being exploited by attackers. Moreover, modern attackers take advantage of zero-day exploits to adversely affect the system, data, or network as long as the vulnerability is not fixed or mitigated. Large and critical organizations are adopting the practice of hiring red teams, a team of security professionals who act as adversaries (Smith et al., 2020), to improve the security of their systems.

Although alternative vulnerability detection tools do exist, e.g., SonarQube (García-Munoz et al., 2016), a static code analysis platform for continuous inspection of code to detect bugs, vulnerabilities, and code smells, and Sensei (De Cremer et al., 2020) that tries to enforce secure coding guidelines in the integrated development environment, it is still very difficult for developers, if not impossible, to build software without vulnerabilities. A key problem is that both static analysis and penetration testing tools have limitations and their low effectiveness in detecting vulnerabilities has been shown in several studies (Herter et al., 2019; Al Shebli and Beheshti, 2018; Scandariato et al., 2013; Araujo Neto and Vieira, 2013).

### 2.4. Software metrics and code trustworthiness

Although we can find several works in the literature that use machine learning algorithms combined with software metrics to detect vulnerable code (Karim et al., 2017; Shen, 2018) including our own work (Medeiros et al., 2020), their results show that such approach is not really effective specially when there is a lack of

resources (e.g., time, money, and human resources) to deal with the large number of false alarms produced. This way, our goal is not to predict/detect vulnerabilities, but instead to propose an approach able to identify the parts of the code that seem to be more untrustworthy, helping developers to review the parts of the code that are more prone to be vulnerable.

In a previous work (Medeiros et al., 2018), we proposed a trustworthiness model that was directly build over a group of weighted software metrics (the weight of each software metric was calculated based on the rank given by a machine learning based prediction model). Despite the promising results, the approach cannot be easily generalized as it is extremely difficult to find a meaningful universal ranking of software metrics based on their importance regarding the prediction models to be used for calculating the score.

We have previously proposed a consensus-based decision-making approach built on top of several machine learning-based prediction models, trained using data on software metrics to categorize code units concerning their security (Medeiros et al., 2021). The results showed that, although software metrics do not constitute sufficient evidence of security issues and cannot effectively be used to build a prediction model to distinguish vulnerable from non-vulnerable code, with a consensus-based decision-making approach it is possible to classify code units from a security perspective. This allows developers to decide which parts of the software should be the focal point for the detection and removal of security vulnerabilities. The main limitation of this approach is that it is mainly based on software metrics and machine learning algorithms and cannot be generalized to other evidence and trustworthiness models.

For the above reasons, in this study, we propose the use of several scores given by diverse scoring models (not necessarily based on machine learning algorithms) for categorizing code. We intend to **build code categorization models based on the trustworthiness scores computed by several machine learning algorithms that are trained using several software metrics**. This can be used to warn developers about the units of code that seem to be more untrustworthy helping them to decide which parts of the software should be the focus of a detailed reviewing process.

## 3. Overall approach

The main idea behind our approach is to use evidences of code security (e.g., software metrics, code smells) not for directly predicting or detecting vulnerabilities but for categorizing each code unit from a security perspective, based on various trustworthiness scores given by several trustworthiness models (or scoring models). In this context, a category refers to the extent to which a piece of code is prone to be untrustworthy (or vulnerable), which varies from *Absolutely Untrustworthy* to *Highly Trustworthy* (the number of categories applied depends on the extent to which we need to characterize the code units in terms of trustworthiness, as will be discussed later).

As shown in Fig. 1, each code unit (e.g., function/method, file/class) is assessed by N Trustworthiness Models ($TM_1$ to $TM_N$), and is assigned with N scores ($S_1$ to $S_N$). The trustworthiness models can be built based on diverse approaches, techniques or tools (in this study, we use several machine learning algorithms to build our TMs) using different evidences of security issues in code (in this work, well known software metrics are used as evidences to train the machine learning algorithms). Each TM gives one score to each code unit. After calculating all scores, the code is classified into different categories representing its trustworthiness (or security) level.

Our approach is based on building a trustworthiness model, not directly on top of evidences (software metrics in this study),

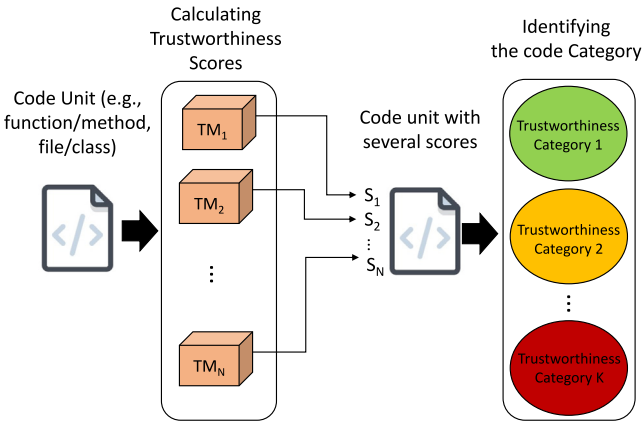


**Fig. 1.** Approach proposed for code categorization from a security perspective.

but instead on the scoring models created over software security evidence (in this work, the models are built using several machine learning algorithms that are trained using software metrics). This solution does not find vulnerable code but can instead warn the developers about untrustworthy (insecure) code units. By identifying the code trustworthiness category, it is up to the developers to decide how many categories of code should be the focus of improvement (e.g., through reviewing, testing, refactoring) depending on the criticality of the application and the available resources, thus, being suitable for any application scenarios.

In order to apply the proposed approach, it is required to build several trustworthiness models that can assign a score to each code unit (files/classes, functions/methods), indicating its trustworthiness level based on evidences observed in the source code.

## 4. Approach instantiation

This section presents a concrete instantiation of the aforementioned approach, taking software metrics as evidences and using Machine Learning algorithms to provide relative importance of each software metric required for building the trustworthiness models.

As shown in Fig. 2, the very first step of this process consists in **preparing (or finding) a dataset** with detailed information about the source code of representative software projects with software metrics and known vulnerabilities.

The second step is to **assign each code unit with several trustworthiness scores**. This requires building several trustworthiness models (TMs), each one outputting one score for each code unit (i.e., 5 TMs are built and 5 scores are assigned to each code unit in this instantiation). To create such a model in this instantiation, we need the normalized value of software metrics (the values will be normalized after being retrieved from the dataset) and their relative weight. To obtain these weights for each TM, several machine learning-based algorithms are used (i.e., 5 ML algorithm, one for each TM). Each machine learning algorithm is used to build one prediction model (different from TMs), which is trained using the aforementioned dataset to predict the vulnerable code based on the structural information of each code unit represented as software metrics. As a result, each prediction model can reveal how important each software metric was in predicting vulnerable code. This information is used to calculate the software metrics relative weights for each TM.

Then, the trustworthiness models (TMs) are built using **Simple Additive Weighting (SAW)** method, which calculated the score of each code unit as the weighted sum of the values of software

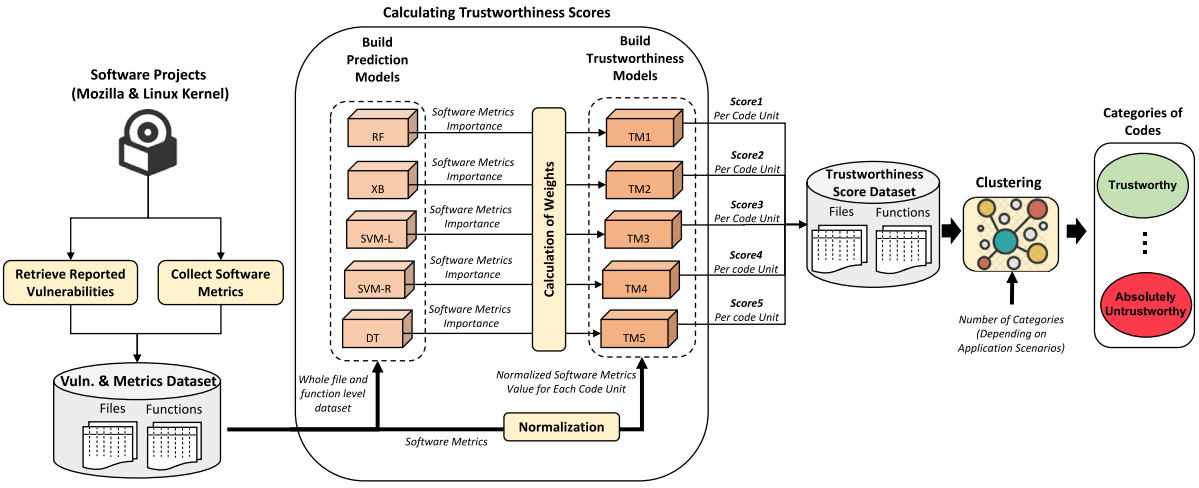


**Fig. 2.** Methodology for approach instantiation.

metrics (Velasquez and Hester, 2013). Although the prediction results of prediction models could be used directly to score the code units, by giving a binary score of 0 or 1 (the predictions models can classify the code units into two classes of vulnerable and non-vulnerable), we do not recommend it as the binary score cannot reflect the difference between the code units in term of the trustworthiness level of properly.

Finally, the last step is focused on **clustering the code** to categorize the code units into different groups with different trustworthiness levels. The following sections describe each step in detail.

It is worth mentioning that although the instantiation of the approach in this study is done by machine learning algorithms and software metrics as evidences, it is not limited to that. Any other trustworthiness and scoring models and security evidences can be applied in the proposed approach. Scoring models created over the results of static code analyzers could be one example of a trustworthiness model. Code smells, and lack of best practices could be other examples of security evidences. Also note that, it is possible to use the classification models as scoring models. To do so, the confidence level (i.e., a probabilistic score between 0 to 1 that represents the likelihood that the classification output – either vulnerable or non-vulnerable in this context – is correct) of the classification result can be used as the score for each code unit. However, these values do not reflect the actual trustworthiness level and cannot be used directly (as both vulnerable and non-vulnerable code units that are classified correctly with high confidence have a high value for the confidence level), therefore must be converted into a score representing the level of trustworthiness of a code unit. In our instantiation, we use classification models to identify the importance of software metrics. Such importance could also be calculated differently, for instance, through statistical analysis, but our solution is more generic (as it is independent of the technique or security evidence for scoring the code units) and is not highly dependent on classification models (classification models are used for scoring the code units and could be replaced with any other scoring models).

### 4.1. Dataset of vulnerabilities and software metrics

In this study, we used a dataset that contains software metrics collected from the source code of five representative software projects from a security perspective (i.e., Mozilla Firefox, Linux Kernel, Apache HTTPd, Xen, and Glibc) as they are all extensively used worldwide and were frequently targeted by attackers. This dataset was created by Henrique Alves (2016) and includes 51 file-level metrics and 28 function-level metrics collected from files and functions of several versions of the five software projects mentioned previously. The metrics were collected using the SciTools Understand tool (SciTools, 2017) and are of different types, including complexity metrics (e.g., Cyclomatic Complexity), volume metrics (e.g., Lines of Code), coupling metrics (e.g., Coupling Between Objects), and cohesion metrics (e.g., Lack of Cohesion). The complete list of software metrics and a short description of each is presented in Section 5.1.1 for both files and functions.

In addition to software metrics, the dataset also contains detailed information about the reported/known vulnerabilities (reported from 2000 to 2016) of the aforementioned software projects. The reported vulnerabilities are collected through analyzing a large number of security patches gathered from two main sources: CVEDetails[2] and Mozilla Foundation Security Advisories (MFSA).[3]

For the current instantiation, we only use two of the projects from the dataset: Linux Kernel (kernel.org) and Mozilla Firefox (mozilla.org). There are two main reasons for this: (i) both are long duration projects with a large codebase, and (ii) both have a considerably high number of reported vulnerabilities (compared to the other three projects in the dataset). Table 1 presents a summary of the data for two projects. The Linux Kernel dataset includes 95,905 files, of which 2,178 have at least one known vulnerability, and 477,342 functions, of which 755 are vulnerable. The Mozilla Firefox dataset consists of 46,444 files and 354,479 functions, of which 844 and 698 are vulnerable, respectively.

### 4.2. Calculating trustworthiness scores

In this work, we build five trustworthiness models (TMs), thus assigning five scores to each unit of code under analysis. For each model, we need the relative importance (weight) of each software metric for each TM and the normalized values of the software metrics.

#### 4.2.1. Building prediction models to weight software metrics

In our previous studies (Medeiros et al., 2017a, 2020), we performed a comprehensive experiment to understand the correlation between software metrics and security vulnerabilities and

[2] CVE Details is a free CVE security vulnerability database/information source. Available on https://www.cvedetails.com.

[3] Mozilla Foundation Security Advisories list security vulnerabilities known to affect particular versions of Mozilla products and instructions on what users can do to protect themselves. Available on https://www.mozilla.org/en-US/security/advisories/.

**Table 1**
Summary of the dataset.

| Software project | # Files | | | # Functions | | |
|---|---|---|---|---|---|---|
| | Total | Non vulnerable | Vulnerable | Total | Non vulnerable | Vulnerable |
| Linux Kernel (C) | 95,905 | 93,727 | 2,178 | 477,342 | 476,587 | 755 |
| Mozilla Firefox (C++) | 46,444 | 45,600 | 844 | 354,479 | 353,781 | 698 |

find out how effective software metrics can be in distinguishing the vulnerable code units from the non-vulnerable ones. We used several statistical techniques and a genetic algorithm in Medeiros et al. (2017a) to find the most correlated software metrics with code security. We also used several machine learning algorithms (Random Forest, Extreme Boosting, Decision Tree, SVM Linear, and SVM Radial) in Medeiros et al. (2020) to extract vulnerability-related knowledge from the same dataset used in this study. In the later study, we also considered different combinations of software metrics and diverse application scenarios with different security concerns (e.g., highly critical or non-critical systems).

According to the results obtained from these studies, we conclude that there is no confident evidence to support that there is a solid and meaningful correlation between specific software metrics and security vulnerabilities. In addition to that, we also observed that the ranking and importance given to software metrics differ among different prediction models. Moreover, no specific combination of software metrics led to the best performance in all classification models. Indeed, the combination of metrics that led to the best performance strongly depends on the machine learning algorithm. For example, Xboost achieved the best result when all metrics were used, while Decision Tree showed a better performance when the redundant metrics were eliminated. Thus, we concluded that feature selection or dimension reduction (selecting a group of software metrics as features) does not help achieve better performance in models created over software metrics to indicate vulnerable code. All these led us to, instead of selecting a group of software metrics, use a large (and relatively complete) set of software metrics of different types to build such models.

To obtain the weights of software metrics to feed each TM, we used the importance given to the software metrics by different machine learning (ML) based prediction models. Each prediction model is built by training a machine learning algorithm for identifying vulnerable code using the software metrics information. Thus, the value given to each software metric by each prediction model indicates its relative importance when used to predict vulnerable code.

We selected five commonly used machine learning algorithms: (i) Decision Tree, (ii) Random Forest, (iii) Linear Support Vector Machine, (iv) Radial Support Vector Machine, and (v) Extreme Gradient Boosted. Decision Tree (DT) is a wildly used algorithm that uses a tree-like model to break up a complex decision problem into several more minor and more straightforward decisions (Safavian et al., 1991). Random forest (RF) is an ensemble method that is made up of a large number of small decision trees (Breiman, 2001). Support Vector Machine (SVM) is a supervised learning algorithm that is used for both classification and regression problems. A classification model built based on the Linear SVM does the classification by identifying a straight line that best separates the target classes by maximizing the margin between them (Awad and Khanna, 2015). In contrast, the Radial SVM is used when there is a nonlinear pattern in the data (Boser et al., 1992). Finally, Extreme Gradient Boosted (Xboost) uses more precise approximations compared to the random forest, as it builds one tree at a time and tries to correct errors made by the previously trained tree in the new one (Chen and Guestrin, 2016; Schapire, 2002).

To configure and run the above algorithms, we used the R Project (Team, 2017) and the R Caret package (Kuhn, 2016). The caret package provides a set of functions that streamline the process of creating predictive models. The package contains different functionalities, such as tools for data splitting pre-processing, feature selection and model tuning using resampling variable importance estimation. The features (software metrics in this study) importance was obtained using the *varImp()* function for each classifier. This function tracks the changes in model performance and other statistics. When a feature (software metrics in this study) is added to the model, it calculates and accumulates the reduction in the performance and statistics. This total reduction is used as a measure for the feature importance.

The machine learning algorithms were tuned to achieve the best prediction result. In the case of Xboost, Linear and Radial SVM, a list of values (based on the literature) were given to the algorithms for each parameter in order to try different combinations, and the best result was selected in each case (e.g., different values for 'nrounds' and 'max_depth' were given for xboost). Also, in order to avoid overfitting and achieve a fair estimation of the performance for each model, we used an internal 10-fold cross-validation and an external 4-fold cross-validation. In practice, we divided the whole dataset into 4 folds. Each ML algorithm was executed four times, and each time it used one fold for testing and 3 folds for training, which internally used a 10-fold cross-validation. The final performance estimation of each classification model is an average of the four estimations.

As a result, we obtain five sets of values for file-level metrics and five sets of values for function-level metrics representing the relative importance of each metric in each vulnerability prediction model. The values for the metrics importance were scaled to have a maximum of 100. Then, each weight was calculated by normalizing the value of the corresponding importance score to a [0–1] range by considering the number of software metrics, so that, for each score, the sum of the weights of all software metrics is 1 (see Tables 3 and 5).

#### 4.2.2. Software metrics normalization

The values of software metrics are scaled differently, so we need to normalize them to a common scale in order to calculate the trustworthiness score (Fig. 2). Our statistical analysis of the distribution of the values on each metric shows cases with very large values for some metrics that affect the normalization process (e.g., the maximum value for FanIn metric in the dataset is 129882 while its mean value is 108). This led us to eliminate the outliers. To do so, we used a statistical method based on the Interquartile Range (IQR), which is defined as the range between the first and the third quartiles (Q3–Q1). Based on IQR, lower fence (LF) and upper fence (UF) are calculated, as detailed in Eq. (1). The Upper and lower fences are defined to set boundaries in data for cordoning off outliers from the dataset. In fact, the acceptable range for the values is defined based on LF and UF. This way, any value falling outside of the acceptable range (i.e., between the LF and UF) is considered to be an outlier.

$$\begin{aligned} &Acceptable\ Range = [Lower\ Fence, Upper\ Fence] \\ &Upper\ Fence(UF) = Q_3 + 1.5 * IQR \\ &Lower\ Fence(LF) = Max(Q_1 - 1.5 * IQR, 0) \\ &IQR = Q_3 - Q_1 \end{aligned} \tag{1}$$

**Feature Scaling**, a known method in data prepossessing (Alshaher, 2021), is used to normalize the values of software metrics and bring them into a common range (between 0 and 1 in our work). **Feature Scaling** is a popular method to normalize the values of independent variables. Our correlation analysis between the software metrics and the existence of security vulnerabilities in code, which is performed in a previous work (Medeiros et al., 2017a), shows an inverse relationship between the software metrics and code security: the greater the value of the metric, the more likely to have a vulnerability, thus less secure. We did not see a clear direct or inverse relationship in a few cases of software metrics. This implies that the selected software metrics have either an inverse relationship or an indifferent relationship. Based on that, the value X for each software metric is scaled into the range [0,1] by subtracting its normalized value from 1 using Eq. (2).

$$X' = \begin{cases} 1 - \dfrac{X - LF}{UF - LF} & : X \text{ is in acceptable range} \\ 0 & : X \text{ is an outlier } (> \text{UF}) \end{cases} \quad (2)$$

*4.2.3. Trustworthiness models*

To build each trustworthiness model (TM) we used **Simple Additive Weighting (SAW)** method. SAW is a commonly used Multi-Criteria Decision-Making method (Aruldoss et al., 2013). According to the SAW method, the score of each given alternative (i.e., a code unit in this context) is calculated as the weighted sum of the quality of each alternative (i.e., the code security in this context) on each attribute (i.e., software metrics in this context) (Velasquez and Hester, 2013). For example, let us assume several functions for being reviewed from a security perspective. Reviewers should evaluate the trustworthiness of each function and assign it a score (in the range of 0 to 1). The evaluation should be performed based on two metrics, e.g., function visibility and number of input parameters, with different importance (e.g., function visibility composes 80% of the final score and the number of input parameters composes 20% of the final score). As a result, each function receives a final trustworthiness score from each reviewer. For one function, if a given reviewer gives 0.9 out of 1 for the first metric and 0.5 out of 1 for the second metric, the final trustworthiness score of the function, calculated as the weighted sum of the two metrics, would be $0.9 * 0.8 + 0.5 * 0.2 = 0.82$.

Based on this concept, to obtain the trustworthiness score for each piece of code (file or function), each TM calculates the trustworthiness score by computing the sum of the product between the (normalized) value of each software metric (M) and its associated weight (W), as shown in Eq. (3), where index $n$ represents the total number of software metrics used.

$$Trustworthiness\ Score = \sum_{i=1}^{n} M_i W_i \quad (3)$$

The difference between the five TMs lies in the weighting system used to identify the importance of the software metrics and the respective weighting. Note that, the score calculated through this TM is a relative measure of trustworthiness that should only be used for comparison purposes and not as an absolute measure of security or trustworthiness.

### 4.3. Clustering the code

The last phase of the methodology is focused on the identification of different groups of code units with the (more or less) same level of trustworthiness perceived from the five different trustworthiness scores. Since there is no known set of rules, patterns, or trained models to transform the vector of scores into a trustworthiness category, we use a **clustering** technique running on the **trustworthiness score dataset** that resulted from the previous step, to characterize and categorize the code units into different groups. Clustering is the task of dividing the data into a certain number of clusters in such a manner that the data belonging to a cluster have similar characteristics (Rai and Singh, 2010). In this work, the clusters are partitioned based on the characteristics of the code units in terms of the trustworthiness score.

There are several techniques to perform clustering, which can be categorized into different types such as, Partitioning techniques, Hierarchical techniques, and Density-based techniques (Saket and Pandya, 2016). The clustering algorithms, in general, follow an iterative process to reassign the data between clusters based on the distance between the clusters (Barioni et al., 2014). In this work, we use **Partitioning Clustering**, which is one of the most commonly used techniques in the literature.

There are different types of partitioning clustering methods. The most popular one is the K-means clustering (MacQueen, 1967), in which each cluster is represented by the center or means of the data belonging to the cluster. However, the K-means method is sensitive to outliers. An alternative to K-means clustering is the K-medoids clustering or Partitioning Around Medoids (PAM) (Kaufman and Rousseeuw, 1990), which is less sensitive to outliers compared to K-means. Clustering Large Applications (CLARA) (Gupta and Panda, 2019) is an extension to the PAM algorithm where the computation time has been reduced to make it perform better for large datasets. Since we have large dataset, we decided to use CLARA algorithm. One important input parameter of clustering algorithms is the number of clusters. In this work, we started by clustering the code units into five clusters, a reasonable number based on our previous study (Medeiros et al., 2021). However, we also considered other values for the number of clusters (3 and 7 clusters) for the sake of comparison.

## 5. Results and discussion

In this section, we present and analyze the results obtained during the instantiation of the approach. The whole dataset used in this study for instantiation and validation, details of all analysis performed, and the obtained results are available online.[4]

### 5.1. Trustworthiness models

To build the trustworthiness models for the calculation of the scores, we first obtained the weights of the software metrics and normalized their values. The list of file-level metrics and function-level metrics and a short description of each are presented in Tables 2 and 4.

*5.1.1. Weight of software metrics*

As mentioned before, we used five machine learning algorithms (Random Forest, Decision Tree, Extreme Gradient Boosted, and Linear and Radial Support Vector Machine) in order to compute the relative importance of software metrics for both file and function level metrics of Linux Kernel and Mozilla Firefox projects. The relative importance was then normalized into the [0,1] range. The final results are presented in Tables 3 and 5 for file-level and function-level, respectively. All 51 file-level metrics and 28 function-level metrics are listed. The results show that the importance of metrics might be different in different prediction models. The values calculated using RF, DT, Xboost, and SVM are different from each other; however, they are the same in the

[4] The dataset used in this study, the obtained results and detailed analysis of the results are available at https://github.com/nadiapsm/Access-2022.

**Table 2**
File-level metrics.

| Software metrics | Description |
|---|---|
| SumEssential | Sum of essential complexity of all nested functions |
| MaxEssential | Maximum essential complexity of all nested functions |
| SumCyclomaticStrict | Sum of strict cyclomatic complexity of all nested functions |
| CountStmtExe | Number of executable statements |
| SumCyclomatic | Sum of cyclomatic complexity of all nested functions |
| CountLineCodeExe | Number of lines containing executable source code |
| AltCountLineComment | Number of lines containing comment, including inactive regions |
| CountLineCode | Number of lines containing source code |
| AltCountLineBlank | Number of blank lines, including inactive regions |
| CountLineBlank | Number of blank lines |
| AvgEssential | Average Essential complexity for all nested functions |
| CountLine | Number of all lines |
| MaxCyclomaticModified | Maximum modified cyclomatic complexity of nested functions |
| CountStmt | Number of statements |
| CountLinePreprocessor | Number of preprocessor lines |
| MaxCyclomaticStrict | Maximum strict cyclomatic complexity of nested functions |
| AltCountLineCode | Number of lines containing source code, including inactive regions |
| SumCyclomaticModified | Sum of modified cyclomatic complexity of all nested functions |
| CountDeclFunction | Number of functions |
| CountLineInactive | Number of inactive lines |
| CountSemicolon | Number of semicolons |
| CountLineComment | Number of lines containing comment |
| MaxCyclomatic | Maximum cyclomatic complexity of all nested functions |
| CountLineCodeDecl | Number of lines containing declarative source code |
| RatioCommentToCode | Ratio of comment lines to code lines |
| CountStmtDecl | Number of declarative statements |
| AvgLine | Average number of lines for all nested functions |
| CountStmtEmpty | Number of empty statements |
| AvgCyclomaticStrict | Average strict cyclomatic complexity for all nested functions |
| AvgLineCode | Average number of lines containing source code for all nested functions |
| MaxFanIn | Maximum number of calling subprograms plus global variables read |
| AltAvgLineCode | Average number of lines containing source code for all nested functions, including inactive regions |
| AvgFanIn | Average number of calling subprograms plus global variables read |
| MaxFanOut | Maximum number of called subprograms plus global variables set |
| CountPath | Number of possible paths, not counting abnormal exits |
| AltAvgLineComment | Average number of lines containing comment for all nested functions, including inactive regions |
| AvgLineBlank | Average number of blank for all nested functions |
| AvgLineComment | Average number of lines containing comment for all nested functions |
| HK | HK measures information flow relative to function size |
| AltAvgLineBlank | Average number of blank lines for all nested functions, including inactive regions |
| MaxNesting | Maximum nesting level of control constructs |
| FanIn | Number of calling subprograms plus global variables read |
| FanOut | Number of called subprograms plus global variables set |
| AvgFanOut | Average number of called subprograms plus global variables set |
| AvgMaxNesting | Average of maximum nesting level of control constructs |
| SumMaxNesting | Sum of maximum nesting level of control constructs |
| AvgCyclomaticModified | Average modified cyclomatic complexity for all nested functions |
| AvgCyclomatic | Average cyclomatic complexity for all nested functions |
| MaxMaxNesting | Maximum nesting level of control constructs |
| CBO | Coupling Between Objects |
| LCOM | Lack of Cohesion in Methods (100% minus the average cohesion) |

case of Linear and Radial SVM; thus, they are presented in a single column (*SVM*). As we can see in the tables, the relative importance of software metrics varies from one prediction model to another. This observation strongly supports the idea behind our proposed approach that we need an integration of several prediction models to make a more precise decision about a piece of code (whether it is untrustworthy or not).

It is worth noting that there are tools like LIME (Ribeiro et al., 2016) that can help to analyze and interpret the results of the prediction models in a more precise way. However, in this work, the prediction models are used as an example for building scoring models (to obtain the weight of software metrics). For this reason, a detailed analysis of the results of the prediction models is out of the scope of the paper and thus left for future work.

### 5.1.2. Normalized values of software metrics

The software metrics value for both files and functions of Linux Kernel and Mozilla Firefox projects were normalized into a common range (between 0 and 1). Table 6 presents examples of the real and normalized values for 4 out of 51 file-level metrics for five files of which three have known vulnerabilities (labeled with **v**), and two do not have known vulnerabilities (labeled with **nv**), as we can see in *Label* column.

As shown in Table 6(b), when the normalized value is 0 that means that the original value is higher than the upper fence and is considered an outlier (e.g., the value of *HK* for files 2, 4, and 5). On the other hand, when the normalized value is 1, it means that the original value is always 0 (e.g., the value of *CountLineInactive* for file 2). For the remaining cases, Eq. (2) is used.

### 5.1.3. Trustworthiness scores for files and functions

For each file and function of each project we obtained five scores (weights of Linear and Radial SVM are equal). An example of the results using files of the Linux Kernel project is presented in Table 7.

As we are interested in characterizing trustworthiness, a higher score should represent a more trustworthy code unit. Accordingly, the example files are ordered from more trustworthy to less trustworthy in the table. Interestingly, the order of the files is the same considering the different scores. Only the third file

**Table 3**
File-level metrics Weight.

| Software metrics | (a) Linux Kernel | | | | (b) Mozilla Firefox | | | |
|---|---|---|---|---|---|---|---|---|
| | RF | DT | Xboost | SVM | RF | DT | Xboost | SVM |
| SumEssential | 0.032 | 0.215 | 0.167 | 0.059 | 0.022 | 0.200 | 0.095 | 0.052 |
| MaxEssential | 0.030 | 0.219 | 0.120 | 0.058 | 0.022 | 0.000 | 0.000 | 0.045 |
| SumCyclomaticStrict | 0.023 | 0.181 | 0.065 | 0.054 | 0.020 | 0.000 | 0.000 | 0.051 |
| CountStmtExe | 0.021 | 0.168 | 0.029 | 0.053 | 0.018 | 0.000 | 0.000 | 0.049 |
| SumCyclomatic | 0.018 | 0.169 | 0.000 | 0.053 | 0.018 | 0.000 | 0.000 | 0.050 |
| CountLineCodeExe | 0.022 | 0.000 | 0.054 | 0.048 | 0.017 | 0.000 | 0.026 | 0.050 |
| AltCountLineComment | 0.025 | 0.000 | 0.051 | 0.044 | 0.021 | 0.200 | 0.058 | 0.050 |
| CountLineCode | 0.022 | 0.000 | 0.040 | 0.048 | 0.020 | 0.000 | 0.052 | 0.051 |
| AltCountLineBlank | 0.022 | 0.000 | 0.037 | 0.047 | 0.020 | 0.201 | 0.091 | 0.052 |
| CountLineBlank | 0.020 | 0.000 | 0.038 | 0.046 | 0.018 | 0.195 | 0.000 | 0.051 |
| AvgEssential | 0.024 | 0.027 | 0.000 | 0.049 | 0.015 | 0.000 | 0.000 | 0.000 |
| CountLine | 0.020 | 0.000 | 0.029 | 0.050 | 0.019 | 0.000 | 0.000 | 0.052 |
| MaxCyclomaticModified | 0.021 | 0.000 | 0.031 | 0.046 | 0.021 | 0.000 | 0.000 | 0.000 |
| CountStmt | 0.017 | 0.000 | 0.028 | 0.051 | 0.015 | 0.000 | 0.000 | 0.050 |
| CountLinePreprocessor | 0.022 | 0.000 | 0.061 | 0.000 | 0.017 | 0.000 | 0.044 | 0.048 |
| MaxCyclomaticStrict | 0.021 | 0.011 | 0.000 | 0.050 | 0.022 | 0.000 | 0.030 | 0.000 |
| AltCountLineCode | 0.022 | 0.000 | 0.000 | 0.050 | 0.018 | 0.000 | 0.029 | 0.052 |
| SumCyclomaticModified | 0.019 | 0.000 | 0.000 | 0.052 | 0.021 | 0.000 | 0.031 | 0.051 |
| CountDeclFunction | 0.026 | 0.000 | 0.000 | 0.044 | 0.020 | 0.000 | 0.030 | 0.047 |
| CountLineInactive | 0.024 | 0.000 | 0.045 | 0.000 | 0.027 | 0.000 | 0.079 | 0.000 |
| CountSemicolon | 0.019 | 0.000 | 0.000 | 0.050 | 0.014 | 0.000 | 0.031 | 0.050 |
| CountLineComment | 0.027 | 0.000 | 0.041 | 0.000 | 0.024 | 0.204 | 0.134 | 0.050 |
| MaxCyclomatic | 0.018 | 0.000 | 0.000 | 0.047 | 0.020 | 0.000 | 0.000 | 0.000 |
| CountLineCodeDecl | 0.026 | 0.000 | 0.040 | 0.000 | 0.016 | 0.000 | 0.026 | 0.051 |
| RatioCommentToCode | 0.026 | 0.000 | 0.033 | 0.000 | 0.028 | 0.000 | 0.047 | 0.000 |
| CountStmtDecl | 0.019 | 0.000 | 0.039 | 0.000 | 0.020 | 0.000 | 0.055 | 0.049 |
| AvgLine | 0.025 | 0.000 | 0.026 | 0.000 | 0.024 | 0.000 | 0.000 | 0.000 |
| CountStmtEmpty | 0.021 | 0.000 | 0.027 | 0.000 | 0.025 | 0.000 | 0.000 | 0.000 |
| AvgCyclomaticStrict | 0.016 | 0.010 | 0.000 | 0.000 | 0.015 | 0.000 | 0.000 | 0.000 |
| AvgLineCode | 0.024 | 0.000 | 0.000 | 0.000 | 0.023 | 0.000 | 0.028 | 0.000 |
| MaxFanIn | 0.023 | 0.000 | 0.000 | 0.000 | 0.019 | 0.000 | 0.000 | 0.000 |
| AltAvgLineCode | 0.023 | 0.000 | 0.000 | 0.000 | 0.024 | 0.000 | 0.000 | 0.000 |
| AvgFanIn | 0.022 | 0.000 | 0.000 | 0.000 | 0.014 | 0.000 | 0.000 | 0.000 |
| MaxFanOut | 0.020 | 0.000 | 0.000 | 0.000 | 0.020 | 0.000 | 0.000 | 0.000 |
| CountPath | 0.019 | 0.000 | 0.000 | 0.000 | 0.024 | 0.000 | 0.000 | 0.000 |
| AltAvgLineComment | 0.019 | 0.000 | 0.000 | 0.000 | 0.016 | 0.000 | 0.000 | 0.000 |
| AvgLineBlank | 0.018 | 0.000 | 0.000 | 0.000 | 0.013 | 0.000 | 0.000 | 0.000 |
| AvgLineComment | 0.018 | 0.000 | 0.000 | 0.000 | 0.017 | 0.000 | 0.000 | 0.000 |
| HK | 0.018 | 0.000 | 0.000 | 0.000 | 0.020 | 0.000 | 0.000 | 0.000 |
| AltAvgLineBlank | 0.018 | 0.000 | 0.000 | 0.000 | 0.013 | 0.000 | 0.000 | 0.000 |
| MaxNesting | 0.017 | 0.000 | 0.000 | 0.000 | 0.018 | 0.000 | 0.000 | 0.000 |
| FanIn | 0.016 | 0.000 | 0.000 | 0.000 | 0.023 | 0.000 | 0.030 | 0.000 |
| FanOut | 0.014 | 0.000 | 0.000 | 0.000 | 0.024 | 0.000 | 0.058 | 0.000 |
| AvgFanOut | 0.013 | 0.000 | 0.000 | 0.000 | 0.020 | 0.000 | 0.000 | 0.000 |
| AvgMaxNesting | 0.013 | 0.000 | 0.000 | 0.000 | 0.017 | 0.000 | 0.000 | 0.000 |
| SumMaxNesting | 0.013 | 0.000 | 0.000 | 0.000 | 0.017 | 0.000 | 0.000 | 0.000 |
| AvgCyclomaticModified | 0.012 | 0.000 | 0.000 | 0.000 | 0.013 | 0.000 | 0.000 | 0.000 |
| AvgCyclomatic | 0.011 | 0.000 | 0.000 | 0.000 | 0.013 | 0.000 | 0.000 | 0.000 |
| MaxMaxNesting | 0.011 | 0.000 | 0.000 | 0.000 | 0.014 | 0.000 | 0.000 | 0.000 |
| CBO | 0.010 | 0.000 | 0.000 | 0.000 | 0.034 | 0.000 | 0.000 | 0.000 |
| LCOM | 0.000 | 0.000 | 0.000 | 0.000 | 0.027 | 0.000 | 0.026 | 0.000 |

(File ID: 15866802) in the case of RF has, a slightly higher score than the second file. However, the scores given to each file have different values. Based on this simple example, we might be able to rank the files based on their combined scores (for instance, by calculating the average value), but that is not easy/possible when the number of files increases and the files do not maintain the same order for all scores.

### 5.2. Clustering results and validation

The output of the trustworthiness models (i.e., scores) is used as input for the clustering process. Thus, for each file and function of both projects, clustering was performed five times: four times based on the individual scores of each model (note that, scores calculated by TMs build using linear SVM and Radial SVM are the same) and one time based on the combination of all scores. The number of clusters in each run is set to 5. The results are presented and discussed in the following subsections.

#### 5.2.1. Clusters of Linux Kernel files and functions

Table 8 presents the results obtained for Linux kernel files (a) and functions (b). The first four columns (i.e., Score RF, Score DT, Score Xboost, and Score SVM) show the results of the clustering performed on each individual score and the last column shows the result of the clustering performed on the combination of all scores. For each cluster, the number of files and function included and the *Medoids* value of their score are also presented. The *Medoids* represents the relative final score of each cluster; thus, it can be used to rank and label the clusters from trustworthy to absolutely untrustworthy (i.e., Cluster 1 is considered as trustworthy and Cluster 5 is considered as absolutely untrustworthy). Interestingly, in all cases, the number of files and functions included in each cluster decreases when the value of *Medoids* decreases. It means that less trustworthy clusters have fewer code units. For instance, 27,156 files (28%) belong to the cluster with a higher level of trustworthiness, and 12,380 files (13%) belong to the cluster with the lowest trustworthiness level (combination of all scores in Table 8(a)).

**Table 4**
Function-level metrics.

| Software Metrics | Description |
|---|---|
| CountOutput | Number of called subprograms plus global variables set |
| CountLineCodeDecl | Number of lines containing declarative source code |
| MaxNesting | Maximum nesting level of control constructs |
| CountInput | Number of calling subprograms plus global variables read |
| AltCountLineBlank | Number of blank lines, including inactive regions |
| Knots | Measure of overlapping jumps |
| CountLineBlank | Number of blank lines |
| CountLineCode | Number of lines containing source code |
| MinEssentialKnots | Minimum Knots after structured programming constructs have been removed |
| AltCountLineComment | Number of lines containing comment, including inactive regions |
| MaxEssentialKnots | Maximum Knots after structured programming constructs have been removed |
| CyclomaticStrict | Strict cyclomatic complexity |
| CountSemicolon | Number of semicolons |
| CountLineComment | Number of lines containing comment |
| CountStmtDecl | Number of declarative statements |
| Cyclomatic | Cyclomatic complexity |
| CountLine | Number of all lines |
| CountLineCodeExe | Number of lines containing executable source code |
| CyclomaticModified | Modified cyclomatic complexity |
| RatioCommentToCode | Ratio of comment lines to code lines |
| CountPath | Number of possible paths, not counting abnormal exits |
| AltCountLineCode | Number of lines containing source code, including inactive regions |
| CountStmtExe | Number of executable statements |
| CountStmt | Number of statements |
| Essential | Essential complexity |
| CountLinePreprocessor | Number of preprocessor lines |
| CountLineInactive | Number of inactive lines |
| CountStmtEmpty | Number of empty statements |

**Table 5**
Function-level metrics Weight.

| Software metrics | (a) Linux Kernel | | | | (b) Mozilla Firefox | | | |
|---|---|---|---|---|---|---|---|---|
| | RF | DT | Xboost | SVM | RF | DT | Xboost | SVM |
| CountOutput | 0.071 | 0.044 | 0.076 | 0.056 | 0.055 | 0.000 | 0.089 | 0.053 |
| CountLineCodeDecl | 0.062 | 0.000 | 0.054 | 0.048 | 0.044 | 0.000 | 0.063 | 0.052 |
| MaxNesting | 0.043 | 0.024 | 0.000 | 0.000 | 0.050 | 0.000 | 0.034 | 0.052 |
| CountInput | 0.042 | 0.039 | 0.068 | 0.050 | 0.042 | 0.000 | 0.077 | 0.000 |
| AltCountLineBlank | 0.040 | 0.004 | 0.050 | 0.049 | 0.045 | 0.207 | 0.065 | 0.055 |
| Knots | 0.040 | 0.000 | 0.047 | 0.051 | 0.025 | 0.000 | 0.026 | 0.040 |
| CountLineBlank | 0.039 | 0.005 | 0.000 | 0.049 | 0.052 | 0.209 | 0.086 | 0.055 |
| CountLineCode | 0.036 | 0.163 | 0.133 | 0.056 | 0.039 | 0.196 | 0.056 | 0.053 |
| MinEssentialKnots | 0.036 | 0.182 | 0.041 | 0.049 | 0.024 | 0.000 | 0.000 | 0.000 |
| AltCountLineComment | 0.036 | 0.005 | 0.031 | 0.000 | 0.037 | 0.000 | 0.029 | 0.045 |
| MaxEssentialKnots | 0.036 | 0.180 | 0.038 | 0.049 | 0.025 | 0.000 | 0.000 | 0.000 |
| CyclomaticStrict | 0.035 | 0.007 | 0.027 | 0.044 | 0.026 | 0.000 | 0.025 | 0.050 |
| CountSemicolon | 0.034 | 0.000 | 0.044 | 0.047 | 0.031 | 0.000 | 0.038 | 0.047 |
| CountLineComment | 0.034 | 0.000 | 0.000 | 0.000 | 0.037 | 0.000 | 0.000 | 0.045 |
| CountStmtDecl | 0.034 | 0.000 | 0.035 | 0.000 | 0.042 | 0.000 | 0.032 | 0.048 |
| Cyclomatic | 0.034 | 0.000 | 0.000 | 0.043 | 0.033 | 0.000 | 0.000 | 0.051 |
| CountLine | 0.033 | 0.156 | 0.059 | 0.056 | 0.042 | 0.196 | 0.103 | 0.055 |
| CountLineCodeExe | 0.033 | 0.000 | 0.045 | 0.056 | 0.035 | 0.000 | 0.050 | 0.051 |
| CyclomaticModified | 0.033 | 0.000 | 0.025 | 0.044 | 0.034 | 0.191 | 0.020 | 0.052 |
| RatioCommentToCode | 0.033 | 0.005 | 0.033 | 0.000 | 0.045 | 0.000 | 0.042 | 0.000 |
| CountPath | 0.033 | 0.000 | 0.049 | 0.050 | 0.024 | 0.000 | 0.045 | 0.049 |
| AltCountLineCode | 0.033 | 0.160 | 0.065 | 0.056 | 0.040 | 0.000 | 0.045 | 0.052 |
| CountStmtExe | 0.032 | 0.000 | 0.039 | 0.048 | 0.040 | 0.000 | 0.040 | 0.047 |
| CountStmt | 0.031 | 0.000 | 0.038 | 0.048 | 0.039 | 0.000 | 0.033 | 0.048 |
| Essential | 0.031 | 0.026 | 0.000 | 0.049 | 0.028 | 0.000 | 0.000 | 0.000 |
| CountLinePreprocessor | 0.021 | 0.000 | 0.000 | 0.000 | 0.007 | 0.000 | 0.000 | 0.000 |
| CountLineInactive | 0.019 | 0.000 | 0.000 | 0.000 | 0.006 | 0.000 | 0.000 | 0.000 |
| CountStmtEmpty | 0.018 | 0.000 | 0.000 | 0.000 | 0.052 | 0.000 | 0.000 | 0.000 |

To study the applicability of the proposed approach and the accuracy of the results, we collected the security vulnerabilities on the Linux Kernel project reported after 2017 (recall that the dataset used in this experiment only contains data from 2000 to 2016). That data was not added to the dataset for training the models, as we wanted to use them for validation. The vulnerabilities were collected from the same sources (CVEDetails). In practice, we identified a total of 801 files and 650 functions with vulnerabilities reported after 2017. The validation consisted of verifying to which cluster each of these files/functions belong to. In other words, for each cluster, we counted the number of unique files/functions with reported vulnerabilities. The assumption is that, if a specific cluster has more reported vulnerabilities than the others, then the cluster (potentially) includes code units more prone to be untrustworthy.

Table 9 presents the results of five clustering experiments, of which four are on individual scores and one is on the combination of all scores. Each table presents the total number of files and the total number of vulnerable files, and their percentages in each cluster. As we can see, the percentage of vulnerable files

**Table 6**
Example of the real (a) and normalized (b) values of four software metrics for five Linux Kernel's files.

(a) File-level dataset of Linux Kernel project

| No. | ID File | Label | Count line inactive | Sum Cyclomatic | Fan Out | HK |
|---|---|---|---|---|---|---|
| **1** | **1407302** | v | 1 | 4 | 12 | 150 |
| **2** | **6081702** | v | 0 | 46 | 164 | 113,172,372 |
| **3** | **15866802** | nv | 141 | 113 | 22 | 13,821,104 |
| **4** | **2627502** | v | 7 | 203 | 270 | 23,115,237 |
| **5** | **32375002** | nv | 302 | 385 | 148 | 1,933,014,887 |

(b) Normalized File-level dataset of Linux Kernel project

| No. | ID File | Label | Count line inactive | Sum Cyclomatic | Fan Out | HK |
|---|---|---|---|---|---|---|
| **1** | **1407302** | v | 0.993 | 0.991 | 0.9 | 1 |
| **2** | **6081702** | v | 1 | 0.898 | 0 | 0 |
| **3** | **15866802** | nv | 0 | 0.750 | 0.817 | 0.126 |
| **4** | **2627502** | v | 0.948 | 0.550 | 0 | 0 |
| **5** | **32375002** | nv | 0 | 0.147 | 0 | 0 |

**Table 7**
Example of trustworthiness scores for five files of Linux Kernel.

| # | ID file | Trustworthiness score | | | |
|---|---|---|---|---|---|
| | | RF | DT | Xboost | SVM |
| **1** | **1407302** | 0.904 | 0.975 | 0.903 | 0.960 |
| **2** | **6081702** | 0.548 | 0.800 | 0.812 | 0.811 |
| **3** | **15866802** | 0.570 | 0.600 | 0.585 | 0.650 |
| **4** | **2627502** | 0.500 | 0.518 | 0.538 | 0.533 |
| **5** | **32375002** | 0.295 | 0.134 | 0.161 | 0.242 |

increases as we move from cluster 1 to cluster 5. For instance, observing the results using the score of RF (Table 9(a)), the less trustworthy cluster (Cluster 5, having a *Medoid* value of 0.304, as shown in Table 8(a)) has the highest percentage of vulnerable files (9.9%). In contrast, the cluster with the highest level of trustworthiness (cluster 1, with a *Medoid* value of 0.85) has the lowest percentage of recently vulnerable files (0.3%). Interestingly, in all cases, the percentage of recent vulnerable files increases when the cluster becomes less trustworthy. The same observation is true in the case of Linux kernel functions presented in Table 10. Indeed, results show that, in fact, the functions categorized as absolutely untrustworthy are more prone to be problematic. Note that having lower values for the percentages in function-level results compared to file-level results is due to the fact that the total number of functions is much higher than the number of files.

A key observation of our study suggested that there is no major difference between the results of clustering on individual scores (considering one single trustworthiness model) and the results of clustering on the combined scores (considering several trustworthiness models by aggregating all scores). This may question the principal idea of our proposed approach by creating doubts on the need to consider various scores when the same clustering results can be achieved using only one score. This is true in the present study as we used the same scoring model in all five models, i.e., the same dataset for creating machine learning-based models built on the same evidence of software security, i.e., the software metrics. Nevertheless, our overall approach is neither limited to machine learning nor to software metrics. Any evidence of security issues and any scoring model can be used to assign different scores to each unit of code.

### 5.2.2. Clusters of Mozilla Firefox files and functions

The clustering results for Mozilla Firefox files and functions are presented in Table 11. The main observations are similar to the ones discussed for Linux Kernel. In fact, the percentage of vulnerable files/functions increases as we move from cluster 1 to cluster 5. Moreover, the percentage of recent vulnerable files/functions increases when the cluster becomes less trustworthy. We again observed that there is no major difference between the results of clustering on individual scores (considering one single trustworthiness model) and the results of clustering on the combined scores (considering several trustworthiness models by aggregating all scores).

To validate the results for Mozilla Firefox results we had to follow a different approach from what was done for the Linux Kernel project. Although we collected the list of reported security vulnerabilities after 2017, the corresponding vulnerable files and functions were not identified in most cases. For this reason, we were not able to use this information. Instead, we used an expert-based ranking to validate those results. In practice, we invited a total of twelve experts with experience and interest in the security area. All invited experts are researchers in secure software engineering, some of whom have more than ten years of experience. The experts are selected from different universities in different countries including Portugal, Italy, Brazil, Belgium. We then selected ten files and ten functions of the Mozilla Firefox project, of which five are labeled as vulnerable and five as non-vulnerable. The procedure for selecting files and functions was as follows:

- **Initial selection:** In order to have a fair ranking, we need to make sure that the files and functions selected are diverse and comparable. To guarantee the diversity in our selection, we first sorted the files/functions according to the value of each software metric. Then, we selected the files/functions with the minimum, lower fence, mean, upper fence, and maximum values for each software metric. This resulted in the selection of a total of 255 files (5 files selected for each of the 51 file-level metrics) and 140 functions (5 functions selected for each of 28 function-level metrics). In the cases where several files/functions had the same value, a random choice among them was made.
- **Cleaning:** In this step, we applied two exclusion criteria: (i) duplicated files/functions must be removed, (ii) the files/functions in which all software metrics have a normalized value equal to 0 must be removed (as in this case, the trustworthiness score cannot be calculated due to division by zero). The step resulted in excluding 184 files and 86 functions.
- **Final selection:** Finally, we chose only 10 files and 10 functions to make the validation process feasible since the experts should compare each pair of files and functions, which is a time-consuming process. To do so, we sorted the remaining 71 files and 54 functions according to their **trustworthiness scores**. Then we selected 5 non-vulnerable files, 5 vulnerable files, 5 non-vulnerable functions, and 5 vulnerable functions with different levels of trustworthiness score ranging from the minimum to maximum.

**Table 8**
Clustering of Linux Kernel files and functions.

(a) File-level results

| Clusters | Score RF | | Score DT | | Score Xboost | | Score SVM | | Score RF, Score DT, Score Xboost, Score SVM | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | # Files | Medoid | # Files | Medoid | # Files | Medoid | # Files | Medoid | # Files | Med. RF | Med. DT | Med. Xboost | Med. SVM |
| Cluster 1 | 22,813 | 0.85 | 28,581 | 0.933 | 24,769 | 0.883 | 26,508 | 0.928 | 27,156 | 0.830 | 0.927 | 0.884 | 0.927 |
| Cluster 2 | 22,793 | 0.758 | 25,838 | 0.801 | 22,591 | 0.795 | 25,721 | 0.801 | 25,098 | 0.738 | 0.815 | 0.759 | 0.802 |
| Cluster 3 | 21,940 | 0.637 | 18,195 | 0.631 | 20,851 | 0.67 | 18,675 | 0.65 | 17,429 | 0.578 | 0.657 | 0.644 | 0.65 |
| Cluster 4 | 15,651 | 0.478 | 12,645 | 0.404 | 15,400 | 0.464 | 13,369 | 0.435 | 13,842 | 0.478 | 0.449 | 0.475 | 0.478 |
| Cluster 5 | 12,708 | 0.304 | 10,646 | 0.076 | 12,294 | 0.182 | 11,632 | 0.136 | 12,380 | 0.337 | 0.117 | 0.155 | 0.148 |
| Total | 95,905 | – | 95,905 | – | 95,905 | – | 95,905 | – | 95,905 | – | – | – | – |

(b) Function-level results

| Clusters | Score RF | | Score DT | | Score Xboost | | Score SVM | | Score RF, Score DT, Score Xboost, Score SVM | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | # Func. | Medoid | # Func. | Medoid | # Func. | Medoid | # Func. | Medoid | # Func. | Med. RF | Med. DT | Med. Xboost | Med. SVM |
| Cluster 1 | 135,767 | 0.885 | 189,464 | 0.928 | 152,339 | 0.930 | 173,559 | 0.921 | 166,562 | 0.876 | 0.935 | 0.923 | 0.924 |
| Cluster 2 | 115,832 | 0.782 | 101,008 | 0.817 | 124,464 | 0.798 | 113,869 | 0.788 | 132,604 | 0.756 | 0.809 | 0.778 | 0.797 |
| Cluster 3 | 95,082 | 0.639 | 79,247 | 0.650 | 88,195 | 0.642 | 78,533 | 0.631 | 83,694 | 0.571 | 0.566 | 0.570 | 0.583 |
| Cluster 4 | 73,553 | 0.448 | 62,324 | 0.351 | 65,060 | 0.405 | 62,298 | 0.405 | 51,796 | 0.386 | 0.286 | 0.326 | 0.343 |
| Cluster 5 | 57,108 | 0.161 | 45,299 | 0.045 | 47,284 | 0.088 | 49,083 | 0.086 | 42,686 | 0.133 | 0.043 | 0.120 | 0.084 |
| Total | 477,342 | – | 477,342 | – | 477,342 | – | 477,342 | – | 477,342 | – | – | – | – |

**Table 9**
Validation of clustering results using files of Linux Kernel project.

(a) Score RF

| Clusters | # Files | % Files | # vuln. | % vuln. |
|---|---|---|---|---|
| Cluster 1 | 22,813 | 23.8% | 79 | 0.3% |
| Cluster 2 | 22,793 | 23.8% | 282 | 1.2% |
| Cluster 3 | 21,940 | 22.9% | 643 | 2.9% |
| Cluster 4 | 15,651 | 16.3% | 855 | 5.5% |
| Cluster 5 | 12,708 | 13.3% | 1,264 | 9.9% |
| Total | 95,905 | - | 3,123 | - |

(b) Score DT

| Clusters | # Files | % Files | # vuln. | % vuln. |
|---|---|---|---|---|
| Cluster 1 | 28,581 | 29.8% | 102 | 0.4% |
| Cluster 2 | 25,838 | 26.9% | 466 | 1.8% |
| Cluster 3 | 18,195 | 19% | 519 | 2.9% |
| Cluster 4 | 12,645 | 13.2% | 850 | 6.7% |
| Cluster 5 | 10,646 | 11.1% | 1,186 | 11.1% |
| Total | 95,905 | - | 3123 | - |

(c) Score Xboost

| Clusters | # Files | % Files | # vuln. | % vuln. |
|---|---|---|---|---|
| Cluster 1 | 24,769 | 25.8% | 82 | 0.3% |
| Cluster 2 | 22,591 | 23.6% | 279 | 1.2% |
| Cluster 3 | 20,851 | 21.7% | 603 | 2.9% |
| Cluster 4 | 15,400 | 16.1% | 816 | 5.3% |
| Cluster 5 | 12,294 | 12.8% | 1,343 | 10.9% |
| Total | 95,905 | - | 3,123 | - |

(d) Score SVM

| Clusters | # Files | % Files | # vuln. | % vuln. |
|---|---|---|---|---|
| Cluster 1 | 26,508 | 27.6% | 66 | 0.2% |
| Cluster 2 | 25,721 | 26.8% | 424 | 1.6% |
| Cluster 3 | 18,675 | 19.5% | 592 | 3.2% |
| Cluster 4 | 13,369 | 13.9% | 820 | 6.1% |
| Cluster 5 | 11,632 | 12.1% | 1,221 | 10.5% |
| Total | 95,905 | - | 3,123 | - |

(e) Score RF, Score DT, Score Xboost, Score SVM

| Clusters | # Files | % Files | # vuln. | % vuln. |
|---|---|---|---|---|
| Cluster 1 | 27,156 | 28.3% | 79 | 0.29% |
| Cluster 2 | 25,098 | 26.2% | 389 | 1.55% |
| Cluster 3 | 17,429 | 18.2% | 555 | 3.18% |
| Cluster 4 | 13,842 | 14.4% | 775 | 5.60% |
| Cluster 5 | 12,380 | 12.9% | 1,325 | 10.7% |
| Total | 95,905 | - | 3,123 | - |

**Table 10**
Validation of clustering results using functions of Linux Kernel project.

(a) Score RF

| Clusters | # Funcs. | % Funcs. | # vuln. | % vuln. |
|---|---|---|---|---|
| Cluster 1 | 135,767 | 28.4% | 37 | 0.03% |
| Cluster 2 | 115,832 | 24.3% | 129 | 0,11% |
| Cluster 3 | 95,082 | 19.9% | 108 | 0,11% |
| Cluster 4 | 73,553 | 15.4% | 153 | 0,21% |
| Cluster 5 | 57,108 | 12% | 175 | 0.31% |
| Total | 477,342 | - | 602 | - |

(b) Score DT

| Clusters | # Funcs. | % Funcs. | # vuln. | % vuln. |
|---|---|---|---|---|
| Cluster 1 | 189,464 | 39.7% | 65 | 0.03% |
| Cluster 2 | 79,247 | 16.6% | 108 | 0.14% |
| Cluster 3 | 101,008 | 21.2% | 160 | 0.16% |
| Cluster 4 | 62,324 | 13.1% | 108 | 0.17% |
| Cluster 5 | 45,299 | 9.5% | 161 | 0.36% |
| Total | 477,342 | - | 602 | - |

(c) Score Xboost

| Clusters | # Funcs. | % Funcs. | # vuln. | % vuln. |
|---|---|---|---|---|
| Cluster 1 | 152,339 | 31.9% | 45 | 0.03% |
| Cluster 2 | 124,464 | 26.1% | 152 | 0.12% |
| Cluster 3 | 88,195 | 18.5% | 113 | 0.13% |
| Cluster 4 | 65,060 | 13.6% | 132 | 0.20% |
| Cluster 5 | 47,284 | 9.9% | 160 | 0.34% |
| Total | 477,342 | - | 602 | - |

(d) Score SVM

| Clusters | # Funcs. | % Funcs. | # vuln. | % vuln. |
|---|---|---|---|---|
| Cluster 1 | 173,559 | 36.4% | 61 | 0.04% |
| Cluster 2 | 78,533 | 16.5% | 89 | 0.11% |
| Cluster 3 | 113,869 | 23.9% | 156 | 0.14% |
| Cluster 4 | 62,298 | 13.1% | 131 | 0.21% |
| Cluster 5 | 49,083 | 10.3% | 165 | 0.34% |
| Total | 477,342 | - | 602 | - |

(e) Score RF, Score DT, Score Xboost, Score SVM

| Clusters | # Funcs. | % Funcs. | # vuln. | % vuln. |
|---|---|---|---|---|
| Cluster 1 | 166,562 | 34.9% | 57 | 0.03% |
| Cluster 2 | 132,604 | 27.8% | 170 | 0.13% |
| Cluster 3 | 83,694 | 17.5% | 127 | 0.15% |
| Cluster 4 | 51,796 | 10.9% | 94 | 0.18% |
| Cluster 5 | 42,686 | 8.9% | 154 | 0.36% |
| Total | 477,342 | - | 602 | - |

After selecting the files and functions, we asked the twelve experts to rank the files and functions based on their perceived trustworthiness by providing them with the source code of the files/functions and the corresponding software metrics. The experts were asked to choose the more trustworthy file/function for each possible pair of code units (files or functions), which resulted in a total of 90 pairwise comparisons. They also were asked to assign a value indicating how much trustworthy a file/function is in comparison to another. For this, we defined and provided them four different levels for comparison: *non-differentiable* (0), *a little more trustworthy* (1), *more trustworthy* (2), or *much more trustworthy* (3). We defined only four levels as we believe that defining more levels does no bring any added value and instead would highly complicate the judgment process for the experts. We then

**Table 11**
Clustering results of Mozilla Firefox project.

(a) File-level results

| Clusters | Score RF | | Score DT | | Score Xboost | | Score SVM | | Score RF, Score DT, Score Xboost, Score SVM | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | # Files | Medoid | # Files | Medoid | # Files | Medoid | # Files | Medoid | # Files | Med. RF | Med. DT | Med. Xboost | Med. SVM |
| Cluster 1 | 14,094 | 0.834 | 19,037 | 0.932 | 16,220 | 0.886 | 16,185 | 0.949 | 15,580 | 0.834 | 0.939 | 0.893 | 0.948 |
| Cluster 2 | 10,991 | 0.724 | 10,314 | 0.804 | 11,638 | 0.778 | 12,069 | 0.833 | 11,595 | 0.716 | 0.848 | 0.795 | 0.827 |
| Cluster 3 | 9,068 | 0.622 | 6,661 | 0.635 | 7,386 | 0.640 | 7,270 | 0.646 | 8,344 | 0.615 | 0.689 | 0.640 | 0.686 |
| Cluster 4 | 7,404 | 0.455 | 4,928 | 0.367 | 5,568 | 0.445 | 5,014 | 0.417 | 5,336 | 0.455 | 0.342 | 0.447 | 0.380 |
| Cluster 5 | 4,887 | 0.256 | 5,504 | 0.000 | 5,632 | 0.148 | 5,906 | 0.038 | 5,589 | 0.307 | 0.000 | 0.147 | 0.043 |
| Total | 46,444 | – | 46,444 | – | 46,444 | – | 46,444 | – | 46,444 | – | – | – | – |

(b) Function-level results

| Clusters | Score RF | | Score DT | | Score Xboost | | Score SVM | | Score RF, Score DT, Score Xboost, Score SVM | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | # Func. | Medoid | # Func. | Medoid | # Func. | Medoid | # Func. | Medoid | # Func. | Med. RF | Med. DT | Med. Xboost | Med. SVM |
| Cluster 1 | 122,468 | 0.833 | 134,553 | 0.954 | 139,925 | 0.884 | 131,761 | 0.940 | 122,019 | 0.844 | 0.954 | 0.900 | 0.949 |
| Cluster 2 | 86,253 | 0.748 | 78,150 | 0.841 | 80,185 | 0.755 | 83,875 | 0.819 | 80,506 | 0.750 | 0.853 | 0.788 | 0.829 |
| Cluster 3 | 59,840 | 0.580 | 60,635 | 0.663 | 54,522 | 0.609 | 56,223 | 0.645 | 54,447 | 0.630 | 0.709 | 0.651 | 0.682 |
| Cluster 4 | 42,818 | 0.389 | 40,298 | 0.369 | 37,772 | 0.374 | 40,862 | 0.403 | 48,038 | 0.424 | 0.507 | 0.462 | 0.516 |
| Cluster 5 | 43,101 | 0.092 | 40,844 | 0.000 | 42,076 | 0.101 | 41,759 | 0.067 | 49,470 | 0.112 | 0.036 | 0.133 | 0.114 |
| Total | 354,480 | – | 354,480 | – | 354,480 | – | 354,480 | – | 354,480 | – | – | – | – |

**Table 12**
Absolute numbers in pairwise comparison.

| Definition | Description | Intensity |
|---|---|---|
| Equal | Non-differentiable | 1 |
| Moderate | Little more trustworthy | 3 |
| Strong | More trustworthy | 5 |
| Very strong | Much more trustworthy | 7 |

transformed all the assigned trustworthiness levels into quantitative values (as shown in Table 12) according to the *Fundamental Scale of Absolute Numbers for Pairwise Comparison* (Martinez et al., 2014). Thus, for supporting further calculations, *non-differentiable* is transformed to 1, *a little more trustworthy* is transformed to 3, *more trustworthy* is transformed to 5, and *much more trustworthy* is transformed to 7.

The first step of the analysis of the experts' responses was focused on the identification of inconsistencies. In this context, an inconsistency occurs when the transitive relation between the files and functions is invalid. For instance, when an expert chooses $F1$ rather than $F2$, $F2$ rather than $F3$, but $F3$ rather than $F1$ as the more trustworthy file/function, while based on the first two statements, $F1$ must be more trustworthy than $F3$. To identify these inconsistencies, we generated a complete graph of the responses, and performed transitivity reduction. We then removed the inconsistent responses from the validation process to eliminate the impact of contradictory responses on the aggregated ranking of files/functions. This resulted in the exclusion of 3 (out of 12) responses/experts (for both files and functions).

In order to perform the aggregated rankings of files and functions based on the trustworthiness perceived by the experts as matrices, we used the Aggregation of Individual Judgments (AIJ) method (Dong et al., 2010). According to this method, the pairwise comparison matrices are aggregated using Geometric Mean. With this, we compared the rankings of files and functions with the clustering results.

The information for files and functions used for validation is presented in Table 13, in which we present the ranking and scores given by experts and compare them with the average *Medoids* of the cluster to which each file belongs. The table is ordered from highest to lowest Average Medoid value and we can clearly observe that the order of files and functions maintains in both cases, except for two functions (ID Function number 121633001 and 119075401) and one file (ID File number 1418301). Observing the source code of those functions and file we noticed that they are very small and simple. We believe that this leads to a higher assessment of trustworthiness in expert based results.

The results obtained clearly suggest that **we were able to achieve a meaningful categorization of the files and functions in terms of trustworthiness using several trustworthiness models.**

### 5.2.3. Considering different number of clusters

The number of files and functions included in the least trustworthy clusters can be high in some application scenarios. Thus, it can be helpful to have a higher number of clusters with less code, allowing the developers to more conveniently chose the groups of code to be worked. For this purpose, we performed an experiment to observe the impact of the number of clusters on the results.

We used the trustworthiness scores obtained using all trustworthiness models for clustering files and functions into 3 and 7 clusters, to be compared with the previously obtained results with 5 clusters. Table 14 presents the results using Linux kernel files (a) and Linux Kernel functions (b). The results obtained (with 3 and 7 clusters) are on par with the previous results (with 5 clusters): the least trustworthy a category (or cluster) is, the higher proportion of the code units included have known vulnerabilities.

We observed that it was possible to categorize the code in a more precise manner by increasing the number of clusters, allowing to achieve a higher intra-cluster similarity and lower inter-cluster similarity. This means that increasing the number of clusters resulted in smaller clusters (with fewer code units) but still with an appropriate (i.e., with respect to the total number of code units in each cluster and considering their trustworthiness level) number of code units with known vulnerabilities (e.g., the least trustworthy category includes a higher percentage of vulnerable code, i.e., 11.6% for files data). Therefore, when there are fewer resources to review code and eliminate vulnerabilities, it is quite useful to perform the clustering with a higher number of clusters. This way, it is possible to decrease the number of code units that must be reviewed and, at the same time, find a reasonable number of vulnerabilities.

To better understand the characteristics of the code units in each cluster, we also calculated the average value of two principal software file and function level metrics: *CountLineCode* and *Cyclomatic* for functions and *CountLineCode* and *SumCyclomatic* for files. Interestingly, in the case of files, the value of both *CountLineCode* and *SumCyclomatic* metrics increases when the level of trustworthiness decreases. This continues to be true even if we increase the number of clusters. In the case of functions, similar results are observed with 5 clusters. However, when the code units are categorized into 3 or 7 clusters, we cannot see big differences between the average values of the two metrics, which implies that other metrics rather than the volume and complexity

**Table 13**
Validation of clustering Results using Mozilla Firefox files and functions.

| ID File | Exp. based results | Average Medoids | ID Function | Exp. based results | Average Medoids |
|---|---|---|---|---|---|
| 60426601 | 0.429 | 0.912 | 361442101 | 0.462 | 0.919 |
| 263601 | 0.408 | 0.912 | 20082901 | 0.458 | 0.919 |
| 1701201 | 0.31 | 0.912 | 304227601 | 0.272 | 0.919 |
| 51355601 | 0.233 | 0.912 | 121633001 | 0.303 | 0.810 |
| 52292101 | 0.165 | 0,803 | 311237201 | 0.153 | 0.671 |
| 1418301 | 0.171 | 0.663 | 113275001 | 0.082 | 0.671 |
| 2201801 | 0.103 | 0,663 | 119075401 | 0.094 | 0.485 |
| 45301201 | 0.07 | 0.108 | 314376601 | 0.057 | 0.485 |
| 22256501 | 0.065 | 0.108 | 328555501 | 0.055 | 0.485 |
| 19960501 | 0.046 | 0.108 | 53225001 | 0.063 | 0.102 |

**Table 14**
Results using different numbers of clusters.

(a) File-level

| | Number of Clusters = 3 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Clusters** | **# Files** | **% Files** | **# Vuln** | **% Vuln** | **CountLineCode** | **SumCyclomatic** | **# Func** | **% Func.** | **# Vuln** | **% Vuln** | **CountLineCode** | **Cyclomatic** |
| Cluster 1 | 43,697 | 45.6% | 288 | 0.7% | 183.3 | 30.7 | 165,580 | 34.7% | 79 | 0.04% | 32.0 | 9.2 |
| Cluster 2 | 33,324 | 34.7% | 1060 | 3.2% | 639.2 | 147 | 138,389 | 29% | 75 | 0.05% | 24.8 | 9.4 |
| Cluster 3 | 18,884 | 19.7% | 1,775 | 9.4% | 2050.5 | 613.2 | 173,373 | 36.3% | 448 | 0.26% | 30.9 | 9.6 |
| Total | 95,905 | – | 3,123 | – | – | – | 477,342 | – | 602 | – | – | – |
| | **Number of Clusters = 5** | | | | | | | | | | | |
| **Clusters** | **# Files** | **% Files** | **# Vuln** | **% Vuln** | **CountLineCode** | **SumCyclomatic** | **# Func** | **% Func.** | **# Vuln** | **% Vuln** | **CountLineCode** | **Cyclomatic** |
| Cluster 1 | 27,156 | 28.3% | 79 | 0.3% | 119.7 | 17.4 | 166,562 | 34.9% | 57 | 0.03% | 6.9 | 1.7 |
| Cluster 2 | 25,098 | 26.2% | 389 | 1.6% | 332.8 | 62.6 | 132,604 | 27.8% | 170 | 0.13% | 17.8 | 5.6 |
| Cluster 3 | 17,429 | 18.2% | 555 | 3.2% | 623.1 | 141.4 | 83,694 | 17.5% | 127 | 0.15% | 32.8 | 10.7 |
| Cluster 4 | 13,842 | 14.4% | 775 | 5.6% | 1073.7 | 277.6 | 51,796 | 10.9% | 94 | 0.18% | 55.9 | 19.3 |
| Cluster 5 | 1,238 | 12.9% | 1,325 | 10.7% | 2479.3 | 764.9 | 42,686 | 8.9% | 154 | 0.36% | 116.2 | 37.0 |
| Total | 95905 | – | 3,123 | – | – | – | 477,342 | – | 602 | – | – | – |
| | **Number of Clusters = 7** | | | | | | | | | | | |
| **Clusters** | **# Files** | **% Files** | **# Vuln** | **% Vuln** | **CountLineCode** | **SumCyclomatic** | **# Func** | **% Func.** | **# Vuln** | **% Vuln** | **CountLineCode** | **Cyclomatic** |
| Cluster 1 | 19,190 | 20% | 46 | 0.2% | 93.4 | 12.0 | 82,066 | 17.2% | 30 | 0.03% | 33.6 | 9.5 |
| Cluster 2 | 18,751 | 19.6% | 168 | 0.9% | 226.5 | 39.4 | 45,003 | 9.4% | 22 | 0.04% | 19.8 | 6.9 |
| Cluster 3 | 19,002 | 19.8% | 397 | 2.09% | 413.2 | 81.7 | 72,538 | 15.2% | 37 | 0.05% | 31.9 | 9.7 |
| Cluster 4 | 13,906 | 14.5% | 459 | 3.3% | 687.5 | 158.6 | 54,978 | 11.5% | 30 | 0.05% | 29.5 | 11.6 |
| Cluster 5 | 9,195 | 9.6% | 459 | 4.99% | 1010.4 | 259.0 | 63,074 | 13.2% | 35 | 0.05% | 25.9 | 8.9 |
| Cluster 6 | 8,367 | 8.7% | 722 | 8.63% | 1520.4 | 463.5 | 80,528 | 16.9% | 166 | 0.21% | 33.7 | 9.8 |
| Cluster 7 | 7,494 | 7.8% | 872 | 11.6% | 3010 | 911.8 | 79,155 | 16.6% | 282 | 0.36% | 27.6 | 9.0 |
| Total | 95,905 | – | 3,123 | – | – | – | 477,342 | – | 602 | – | – | – |

(b) Function-level

metrics (e.g., *MaxNesting*) were influential in the trustworthiness level of functions. However, in most cases, the results show that when the code's size and complexity increase, its trustworthiness level decreases.

Here, we highlight the main insights obtained from the results:

- The relative importance of software metrics varies from one prediction model to another. This strongly supports the idea behind our proposed approach that we need an integration of several scoring models to make a more precise decision about a piece of code.
- The number of files and functions included in each cluster decreases when the trustworthiness level decreases. It means that less trustworthy clusters have fewer code units.
- The percentage of new reported vulnerabilities increases as we move from trustworthy cluster to absolutely untrustworthy cluster. This strongly support the validation of the proposed clustering-based approach. Moreover, the expert based results also validated the effectiveness of our approach.
- Our overall approach is neither limited to machine learning nor to software metrics. Any evidence of security issues and any scoring model can be used to assign different scores to each unit of code.
- We categorized the code units in a more precise manner by increasing the number of clusters, allowing to achieve a higher intra-cluster similarity and lower inter-cluster similarity. This means that increasing the number of clusters resulted in smaller clusters (with fewer code units) with more similar characteristics in terms of software metrics and trustworthiness level. Thus, when there are fewer resources to review code and eliminate vulnerabilities, it is quite useful to perform the clustering with a higher number of clusters. This way, it is possible to decrease the number of code units that must be reviewed and, at the same time, find a reasonable number of vulnerabilities.

## 6. Threats to validity

Although the results show that we can effectively use our approach to make effective decisions about the parts of code that might be problematic and require deeper analysis, some internal and external limitations and threats to validity should be highlighted:

**Internal Limitations and Threats:**

- The prediction models are built using machine learning algorithms and software metrics are used as evidence of security issues in code. Although the instantiation experiment we performed in this study shows the applicability of the proposed approach, as a future work we aim to use different tools and techniques as scoring models as well as other security evidences (e.g., code smells).
- The clustering results obtained from a single score and the combination of several scores are pretty similar. This was expected as, in the present study, we used a similar scoring model in all five models (same dataset for creating machine learning-based models built on the same evidence of software security, i.e., the software metrics). However, our approach is neither limited to machine learning nor software

metrics. Validation of our approach with different scoring models and different pieces of evidence is considered one of the future works.
- The number of files and functions selected to validate the results of Mozilla Firefox project are also limited (10 files and 10 functions). This is an acceptable number of files and functions, since the experts had to perform the comparison between all the possible pairs. However it would be better to get a larger number of compared code units.
- The dataset used for instantiating the proposed approach and its attestation is limited to two projects. Although the projects are quite big (with a large number of files and functions used to build the trustworthiness models) and representative for this study, including more projects could help achieving more convincing results.
- Although the main idea of the paper is to avoid considering only two classes (vulnerable and non-vulnerable) of code units (as it makes the decision-making process more difficult for the developers who need to review the code units), it would be interesting to cluster the code units into two categories of trustworthy and untrustworthy code units to see how the actual vulnerable and non-vulnerable code units are distributed in these two categories, helping to understand whether we can avoid clustering and use the results of several predictors directly to make a decision about the code units. This study can be done in the future.

**External Limitations and Threats:**

- Responses of nine experts (out of twelve as three inconsistent responses were excluded) were considered in order to validate the proposed approach using Mozilla Firefox data. We should get as many responses as possible from experts in software security field, however this is a difficult task. Nevertheless, we consider that the results obtained provide a good basis for demonstrating the validity of the proposed approach.
- To attest to the applicability of the proposed approach, a real development team working on an actual project could be used. Managing and performing such experiments is quite challenging, being one of our future works.

## 7. Conclusions and future work

In this paper, we proposed an approach that aims to identify code units that are more prone to be vulnerable, thus helping developers to improve code security. To do so, we firstly built different trustworthiness models based on several machine learning algorithms that are trained using several software metrics. Then we used the trustworthiness scores as input for the clustering process, where we categorize the code units by the level of trustworthiness.

Results show that we can effectively use clustering in order to identify code units more prone to be vulnerable. Thus, developers can use this approach to make efficient and effective decisions about the parts of code that might be problematic and require deep analysis. Besides that, it is possible to adjust the clustering process considering different scenarios, where different resources to detect and eliminate vulnerabilities are available.

As future work, the approach can be extended to different trustworthiness models and to other types of evidence of security issues (e.g., code smells). The main objective will be to use several evidences of security issues in order to categorize code units into several groups based on the trustworthiness level, supporting developers in the detection of potential issues during the software development process. Moreover, machine learning models could be used as the scoring system in this approach. The confidence level of the classification could be used as the score for each code unit. This idea will be explored in the future.

## CRediT authorship contribution statement

**Nadia Medeiros:** Conception and design of study, Acquisition of data, Analysis and/or interpretation of data, Writing – original draft. **Naghmeh Ivaki:** Conception and design of study, Analysis and/or interpretation of data, Writing – original draft. **Pedro Costa:** Conception and design of study, Writing – review & editing. **Marco Vieira:** Conception and design of study, Writing – review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Data will be made available on request.

## Acknowledgments

This work has been supported by the project "METRICS: Monitoring and Measuring the Trustworthiness of Critical Cloud Systems", funded by the Portuguese Foundation for Science and Technology (FCT) with agreement number POCI-01-0145-FEDER-032504; and by the project "AIDA: Adaptive, Intelligent and Distributed Assurance Platform", funded by the FCT, COMPETE2020, CMU Portugal Program, and the European Regional Development Fund (ERDF).

All authors approved the version of the manuscript to be published.

## References

Al Shebli, H.M.Z., Beheshti, B.D., 2018. A study on penetration testing process and tools. In: 2018 IEEE Long Island Systems, Applications and Technology Conference. LISAT, IEEE, pp. 1–7.

Alarcon, G.M., Gamble, R., Jessup, S.A., Walter, C., Ryan, T.J., Wood, D.W., Calhoun, C.S., 2017a. Application of the heuristic-systematic model to computer code trustworthiness: The influence of reputation and transparency. Cogent Psychol. 4 (1), 1389640.

Alarcon, G.M., Militello, L.G., Ryan, P., Jessup, S.A., Calhoun, C.S., Lyons, J.B., 2017b. A descriptive model of computer code trustworthiness. J. Cogn. Eng. Decis. Mak. 11 (2), 107–121.

Alshaher, H., 2021. Studying the Effects of Feature Scaling in Machine Learning (Ph.D. thesis). North Carolina Agricultural and Technical State University.

Alves, H., Fonseca, B., Antunes, N., 2016. Software metrics and security vulnerabilities: Dataset and exploratory study. In: 12th European Dependable Computing Conference. EDCC, IEEE, pp. 37–44.

Araujo Neto, A., Vieira, M., 2013. Selecting secure web applications using trustworthiness benchmarking. Int. J. Dependable Trustworthy Inf. Syst. 2, 1–16.

Arkin, B., Stender, S., McGraw, G., 2005. Software penetration testing. IEEE Secur. Priv. 3 (1), 84–87.

Aruldoss, M., Lakshmi, T.M., Venkatesan, V.P., 2013. A survey on multi criteria decision making methods and its applications. Am. J. Inf. Syst. 1 (1), [Online]. Available: http://pubs.sciepub.com/ajis/1/1/5.

Awad, M., Khanna, R., 2015. Support vector machines for classification - Efficient learning machines: Theories, concepts, and applications for engineers and system designers. In: Efficient Learning Machines. Apress, pp. 39–66.

Barioni, M., Razente, H., Marcelino, A., Traina, A., Jr., C., 2014. Open issues for partitioning clustering methods: An overview. Wiley Interdiscip. Rev.: Data Min. Knowl. Discov. 4.

Beckers, K., Côté, I., Fenz, S., Hatebur, D., Heisel, M., 2014. A structured comparison of security standards. In: Engineering Secure Future Internet Services and Systems. Springer, pp. 1–34.

Boser, B., Guyon, I., Vapnik, V., 1992. A training algorithm for optimal margin classifiers. In: COLT '92 Proceedings of the Fifth Annual Workshop on Computational Learning Theory. ACM, pp. 144–152.

Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32.

Cavoukian, A., 2009. Privacy by design - the 7 foundational principles - implementation and mapping of fair information practices. Information & Privacy Commissioner, Ontario, Canada.
Chemuturi, M., 2010. Mastering Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers. J. Ross Publishing.
Chen, T., Guestrin, C., 2016. XGBoost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 785–794.
Chess, B., McGraw, G., 2004. Static analysis for security. IEEE Secur. Priv. 2 (6), 76–79.
De Cremer, P., Desmet, N., Madou, M., De Sutter, B., 2020. Sensei: Enforcing secure coding guidelines in the integrated development environment. Softw. - Pract. Exp. 50 (9), 1682–1718.
Del Bianco, V., Lavazza, L., Morasca, S., Taibi, D., 2011. A survey on open source software trustworthiness. IEEE Softw. 28 (5), 67–75.
Deng, R., Chen, Y., Wu, H., Tao, H., 2019. Software trustworthiness evaluation using structural equation modeling. Int. J. Perform. Eng. 15 (10), 2628.
Disterer, G., 2013a. ISO/IEC 27000, 27001 and 27002 for Information Security Management. International Organization for Standardization (ISO).
Disterer, G., 2013b. ISO/IEC 27000, 27001 and 27002 for information security management. J. Inform. Secur. 4.
Dong, Y., Zhang, G., Hong, W.-C., Xu, Y., 2010. Consensus models for AHP group decision making under row geometric mean prioritization method. Decis. Support Syst. 49 (3), 281–289, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167923610000643.
Evans, D., Larochelle, D., 2002. Improving security using extensible lightweight static analysis. IEEE Softw. 19 (1), 42–51.
Galin, D., 2004. Software Quality Assurance: From Theory to Implementation. Pearson Education India.
García-Munoz, J., García-Valls, M., Escribano-Barreno, J., 2016. Improved metrics handling in SonarQube for software quality monitoring. In: Distributed Computing and Artificial Intelligence, 13th International Conference. Springer, pp. 463–470.
Graff, M., Wyk, V., 2003. Secure coding: Principles and practices. In: Designing and Implementing Secure Applications. O'Reilly Media Inc.
Gupta, T., Panda, S.P., 2019. A comparison of K-means clustering algorithm and CLARA clustering algorithm on iris dataset. Int. J. Eng. Technol. 7 (4), [Online]. Available: https://www.sciencepubco.com/index.php/ijet/article/view/21472.
Heimann, D., 2014. IEEE Standard 730-2014 Software Quality Assurance Processes, Vol. 730. IEEE Computer Society, New York, NY, USA, IEEE Std, p. 2014.
Henrique Alves, N.A., 2016. A dataset of source code metrics and vulnerabilities. [Online]. Available: https://eden.dei.uc.pt/nmsa/metrics-dataset/.
Herter, J., Kästner, D., Mallon, C., Wilhelm, R., 2019. Benchmarking static code analyzers. Reliab. Eng. Syst. Saf. 188, 336–346.
Horvath, A.S., Agrawal, R., 2015. Trust in cloud computing. In: SoutheastCon 2015. IEEE, pp. 1–8.
I. ISO and I. Std, 2009. Iso 15408-1: 2009, information technology-security techniques-evaluation criteria for IT security. Part 1.
Jung, H., Kim, S., Chung, C., 2004. Measuring software product quality: A survey of ISO/IEC 9126. IEEE Softw. 21 (5), 88–92.
Karim, S., et al., 2017. Software metrics for fault prediction using machine learning approaches: A literature review with PROMISE repository dataset. In: 2017 IEEE International Conference on Cybernetics and Computational Intelligence. CyberneticsCom, IEEE, pp. 19–23.
Kaufman, L., Rousseeuw, P.J., 1990. Finding Groups in Data: An Introduction to Cluster Analysis. Wiley, New York.
Kuhn, M., 2016. The R Caret Package. [Online]. Available: https://cran.r-project.org/web/packages/caret/index.html.
Lee, L.S., Brink, W.D., 2020. Trust in cloud-based services: A framework for consumer adoption of software as a service. J. Inf. Syst. 34 (2), 65–85.
MacQueen, J., 1967. Some methods for classification and analysis of multivariate observations. In: Proceedings of the Symposium on Mathematics and Probability. pp. 281–297.
Marcil, J., 2014. OWASP ISO IEC 27034 Application Security Controls Project. OWASP-Open Web Application Security Project.
Martinez, M., de Andres, D., Ruiz, J.-C., Friginal, J., 2014. From measures to conclusions using analytic hierarchy process in dependability benchmarking. IEEE Trans. Instrum. Meas. 63 (11), 2548–2556.
Medeiros, N., Ivaki, N., Costa, P., Vieira, M., 2017a. Software metrics as indicators of security vulnerabilities. In: 28th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 216–227.
Medeiros, N., Ivaki, N., Costa, P., Vieira, M., 2018. An approach for trustworthiness benchmarking using software metrics. In: 2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing. PRDC, IEEE, pp. 84–93.
Medeiros, N., Ivaki, N., Costa, P., Vieira, M., 2020. Vulnerable code detection using software metrics and machine learning. IEEE Access.
Medeiros, N., Ivaki, N., Costa, P., Vieira, M., 2021. An empirical study on software metrics and machine learning to identify untrustworthy code. In: 17th European Dependable Computing Conference. EDCC.
Medeiros, N., Ivaki, N.R., Da Costa, P.N., Vieira, M.P.A., 2017b. Towards an approach for trustworthiness assessment of software as a service. In: 2017 IEEE International Conference on Edge Computing. EDGE, IEEE, pp. 220–223.
Neto, A.A., Vieira, M., 2011. To benchmark or not to benchmark security: That is the question. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops. DSN-W, pp. 182–187.
Potii, O., Illiashenko, O., Komin, D., 2015. Advanced security assurance case based on ISO/IEC 15408. In: International Conference on Dependability and Complex Systems. Springer, pp. 391–401.
Poulin, L., Guay, B., 2008. ISO/IEC 27034 application security-overview. p. 29, 20.1 Kyoto.
Qiu, L., Zhang, Y., Wang, F., Kyung, M., Mahajan, H.R., 1985. Trusted Computer System Evaluation Criteria. National Computer Security Center.
Rai, P., Singh, S., 2010. A survey of clustering techniques. Int. J. Comput. Appl. 7 (12).
Rawat, M.S., Mittal, A., Dubey, S.K., 2012. Survey on impact of software metrics on software quality. (IJACSA) Int. J. Adv. Comput. Sci. Appl. 3 (1).
Ribeiro, M.T., Singh, S., Guestrin, C., 2016. "Why should I trust you"? Explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1135–1144.
Safavian, S., et al., 1991. A survey of decision tree classifier methodology. In: IEEE Transactions on Systems, Man, and Cybernetics.
Saket, S., Pandya, S., 2016. An overview of partitioning algorithms in clustering techniques. Int. J. Adv. Res. Comput. Eng. Technol. (IJARCET) 5.
Scandariato, R., Walden, J., Joosen, W., 2013. Static analysis versus penetration testing: A controlled experiment. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 451–460.
Schapire, R., 2002. The boosting approach to machine learning -An overview. In: Nonlinear Estimation and Classification. In: Lecture Notes in Statistics, vol. 171, pp. 149–171.
SciTools, 2017. Understand static code analysis tool. [Online]. Available: https://scitools.com/feature/metrics/.
SCSUG, 2001. Common Criteria for Information Technology Security Evaluation: Smart Card Protection Profile, Vol. 3.0. Ray-McGovern Technical Consultants, Inc..
Seacord, R.C., Householder, A.D., 2005. A Structured Approach to Classifying Security Vulnerabilities. TECHNICAL NOTE: CMU/SEI-2005-TN-003, Carnegie Mellon Software Engineering Institute, [Online]. Available: https://apps.dtic.mil/sti/pdfs/ADA430968.pdf.
Shan, L., Sangchoolie, B., Folkesson, P., Vinter, J., Schoitsch, E., Loiseaux, C., 2019. A survey on the applicability of safety, security and privacy standards in developing dependable systems. In: International Conference on Computer Safety, Reliability, and Security. Springer, pp. 74–86.
Shen, X., 2018. Predicting vulnerable files by using machine learning method. Faculty of Electrical Engineering, Mathematics and Computer Science(EWI), Delft University of Technology.
Smith, J., Theisen, C., Barik, T., 2020. A case study of software security red teams at microsoft. In: 2020 IEEE Symposium on Visual Languages and Human-Centric Computing. VL/HCC, IEEE, pp. 1–10.
Team, R.C., 2017. The R project for statistical computing.
Turpin, K., 2010. OWASP Secure Coding Practices-Quick Reference Guide.
Velasquez, M., Hester, P., 2013. An analysis of multi-criteria decision making methods. Int. J. Oper. Res. 10, 56–66.

**Nádia Medeiros** is a PhD student and researcher of Software and Systems Engineering Group (SSE) of the Centre for Informatics and Systems (CISUC), Department of Informatics Engineering, University of Coimbra. Her research interests include trustworthiness benchmarking and software quality assurance. She has authored several peer-reviewed publications in this field.

**Naghmeh Ivaki** received the PhD degree from the University of Coimbra, Portugal. Currently, she is an invited assistant professor and a full member of Software and Systems Engineering Group (SSE) of the Centre for Informatics and Systems (CISUC), Department of Informatics Engineering, University of Coimbra. She specializes in the scientific field of Informatics Engineering, with particular focus on security and dependability of computer systems. In her field of specialization, she has authored more than 30 peer-reviewed publications and participated in several national and international research projects.

**Pedro Costa** is a computer science professor at Polytechnic Institute of Coimbra and a researcher in Center for Informatics and Systems of the University of Coimbra, where he integrates the Software and Systems Engineering Group (SSE). His interests include dependable systems, dependability and security assessment and benchmarking, software fault tolerance and software implemented fault injection (SWIFI). Pedro Costa is currently the President of Coimbra Business School (ISCAC), Portugal.

**Marco Vieira** is a Full Professor at the University of Coimbra, Portugal. His interests include dependability and security assessment and benchmarking, fault injection, software processes, and software quality assurance, subjects in which he has authored or co-authored more than 200 papers in refereed conferences and journals. He has participated and coordinated several research projects, both at the national and European level. Marco Vieira has served on program committees of the major conferences of the dependability area and acted as referee for many international conferences and journals in the dependability and security areas.