IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 2, 2020, accepted November 21, 2020, date of publication November 27, 2020, date of current version December 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3041181

Vulnerable Code Detection Using Software Metrics and Machine Learning

NÁDIA MEDEIROS[1], NAGHMEH IVAKI[1], (Member, IEEE), PEDRO COSTA[1,2], AND MARCO VIEIRA[1], (Member, IEEE)

[1]Centre for Informatics and Systems (CISUC), Department of Informatics Engineering, University of Coimbra, 3030-290 Coimbra, Portugal
[2]Coimbra Business School, ISCAC, Polytechnic Institute of Coimbra, 3045-093 Coimbra, Portugal

Corresponding author: Naghmeh Ivaki (naghmeh@dei.uc.pt)

This work was supported in part by the project METRICS: Monitoring and Measuring the Trustworthiness of Critical Cloud Systems of the Portuguese Foundation for Science and Technology (FCT) under Grant POCI-01-0145-FEDER-032504, and in part by the project AIDA: Adaptive, Intelligent and Distributed Assurance Platform of the FCT, COMPETE2020, CMU Portugal Program, and the European Regional Development Fund (ERDF) under Grant POCI-01-0247-FEDER-045907.

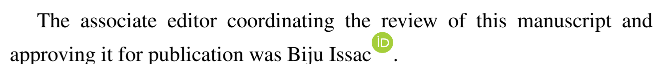
ABSTRACT Software metrics are widely-used indicators of software quality and several studies have shown that such metrics can be used to estimate the presence of vulnerabilities in the code. In this paper, we present a comprehensive experiment to study how effective software metrics can be to distinguish the vulnerable code units from the non-vulnerable ones. To this end, we use several machine learning algorithms (Random Forest, Extreme Boosting, Decision Tree, SVM Linear, and SVM Radial) to extract vulnerability-related knowledge from software metrics collected from the source code of several representative software projects developed in C/C++ (Mozilla Firefox, Linux Kernel, Apache HTTPd, Xen, and Glibc). We consider different combinations of software metrics and diverse application scenarios with different security concerns (e.g., highly critical or non-critical systems). This experiment contributes to understanding whether software metrics can effectively be used to distinguish vulnerable code units in different application scenarios, and how can machine learning algorithms help in this regard. The main observation is that using machine learning algorithms on top of software metrics helps to indicate vulnerable code units with a relatively high level of confidence for security-critical software systems (where the focus is on detecting the maximum number of vulnerabilities, even if false positives are reported), but they are not helpful for low-critical or non-critical systems due to the high number of false positives (that bring an additional development cost frequently not affordable).

INDEX TERMS Application scenarios, machine learning, software metrics, software security, security vulnerabilities.

I. INTRODUCTION

Several research studies show that software defects/vulnerabilities (e.g., Buffer overflow, SQL injection) are a central and critical source of security breaches [1]–[3] in computer systems. Such vulnerabilities are mainly caused by unprofessional or negligent developers who lack security knowledge [4]. To instruct software developers to incorporate security and, in general, quality into software, there are several well-established and widely known standards and best practice recommendations, such as Software Quality Assurance (SQA) [5], Quality by Design (QbD), OWASP secure coding practices, ISO / IEC 27034, and Privacy by Design [6]. However, research and experience show that modern software still fails in meeting basic security requirements [7].

A large number of tools and techniques to detect security vulnerabilities are nowadays available. For example, static code analysis [8] is a well-known technique used by developers to search for software defects and vulnerabilities in early stages of the software development. Nevertheless, detecting software vulnerabilities or distinguishing vulnerable from non-vulnerable code is not trivial. The low effectiveness of vulnerability detection tools and static code analyzers is a clear proof of this fact [9]. Thus, software is often deployed with bugs that can be exploited by attackers causing system outages, data breaches, or even safety issues. This has led to


The associate editor coordinating the review of this manuscript and approving it for publication was Biju Issac.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

many works trying to mitigate the damage that the exploitation of such vulnerabilities can cause at runtime.

In this work, we focus on prevention and on early detection of software vulnerabilities, as fixing those vulnerabilities is easier and less expensive and the consequences are potentially smaller (compared with the detection of vulnerabilities in later development stages or after deployment) [8]. In this direction, we present an empirical study on one of the early evidences of software quality, namely software metrics, whose correlation with the existence of security vulnerabilities has been shown in previous works [10], [11]. **In practice, we aim to understand how the information provided by software metrics can be best used by machine learning algorithms to distinguish the vulnerable code units (files, functions) from the non-vulnerable ones with high levels of confidence within different circumstances, including different application scenarios that encompass diverse security concerns.**

This study considers several commonly used machine learning (ML) algorithms (Random Forest, Extreme Boosting, Decision Tree, SVM Linear and SVM Radial) that are applied on software metrics of all types (e.g., Cyclomatic Complexity, Lines of Code, and Coupling Between Objects) collected from the source code of several widely used and representative software projects developed in C/C++ (Mozilla Firefox, Linux Kernel, Apache HTTPd, Xen and Glibc) at different levels (file level and function level). We consider different combinations of software metrics, selected based on different approaches (e.g., correlation analysis), and focus on four application scenarios (Highly-Critical, Critical, Low-Critical and Non-Critical) that have different concerns regarding security, thus requiring diverse criteria for evaluating the classifiers. For instance, in the highly-critical systems scenario, detection and elimination of vulnerabilities is of high priority even if some false alarms are reported, therefore, a criterion that measures the ratio of detected vulnerable code units independently from false alarms seems to be of interest. In contrast, in the non-critical systems, the number of false alarms can be the main concern due to limited development resources, thus, a criterion that in addition to the correctly classified vulnerable code, strongly rewards low false alarms seems to be adequate.

This work intends to contribute to answer the following research questions (RQs):

- **RQ1**. Can software metrics effectively be used to distinguish vulnerable code units from the non-vulnerable ones in different application scenarios?
- **RQ2**. What is the best combination of software metrics to be used for this purpose?
- **RQ3**. How do different machine learning algorithms perform in this context?
- **RQ4**. Can the results of this experiment be generalised and applied to different types of software systems?

We use the dataset built by Alves *et al.* [12], which includes software metrics and reported security vulnerabilities for all code units (e.g., functions and files) of several versions of different widely used software projects (Mozilla Firefox, Linux Kernel, Apache HTTPd, Xen and Glibc). Results show that the models created over software metrics are effective for security-critical applications (highly-critical and critical), in which the detection of vulnerabilities is of high priority. In contrast, a large number of false alarms make them useless for scenarios with low critical or non-critical systems (where budget to deal with vulnerabilities is limited). This suggests that it is quite important to consider application scenarios when building vulnerability detecting tools. From the analyzed classifiers, Random Forest and Extreme Boosting are the ones that lead to more precise models for both file and function level data. However, Decision Tree and Linear SVM build more generalizable models, thus, giving a better estimation when completely unknown data is used for testing.

The rest of the paper is organized as follows. Section II reviews the related work. Section III explains the approach, methods and techniques used to conduct the experiments and the analysis. A preliminary analysis of the results, focused on defining the configurations and settings to be used in the experiments, is presented in Section IV. Section V presents the results and their analysis. The main outcomes of the paper and the threats to the validity of the work are summarized and discussed in Section VI. Section VII concludes the paper and puts forward ideas for future work.

II. BACKGROUND AND RELATED WORK

Modern businesses, organizations, and critical infrastructures are backed by software systems executing critical operations and transactions, providing services and dealing with huge amounts of sensitive data for supporting effective decisions and constant business/system adaptation. This tremendously increased concerns regarding security, driving researchers and businesses to come up with tools, techniques, standards, and regulations to help developers to ensure security in software systems [13], [14].

We can find a lot of efforts in the literature focused on the definition of best practices, standards, and regulations to help developers in building high quality and secure software (e.g., ISO/IEC 27000 [15], ISO 15408 [16], Software Quality Assurance (SQA) [5], [17], [18], OWASP secure coding practices [19], [20], ISO/IEC 27034 [21], and Privacy by Design (PbD) [6]) [22]–[24]. An structured description and comparison between most of these efforts can be found in [25] and [26].

We also can find many works on tools and techniques to prevent or detect and eliminate software bugs during the software development process [27], [28], like SonarQube [29], a platform for continuous inspection (static analysis) of code to detect bugs, vulnerabilities, and code smells. Sensei [30] is another example that tries to enforce secure coding guidelines in the integrated development environment. However, it is still very difficult for developers, if not impossible, to build software without vulnerabilities. This has led to many works trying to mitigate the damage that such vulnerabilities can cause at runtime (e.g., via via intrusion detection systems and

attack tolerance techniques) [31]–[35]. Despite all existing efforts, software is still shipped with exploitable vulnerabilities causing huge damages to the systems and businesses. Thus, better and more effective approaches to detect vulnerabilities earlier in the life cycle are still needed [36].

Existing approaches for the detection of vulnerabilities in the early stages of the software development can be divided in two categories: static code analysis [8] and penetration testing. In static code analysis, the source code (or compiled code) of a software is examined statically, without executing it. Static analysis of code can be done manually or by using static analysis tools (SATs). Manual auditing of code is time consuming and requires skilled human code auditors with sufficient and deep knowledge regarding security vulnerabilities and security attacks to be able to effectively examine the code. In contrast, static analysis tools encapsulate security knowledge in a way that does not require highly skilled human auditors with security expertise, thus, are faster and can be frequently used examine the code. Nevertheless, the output of these tools still requires evaluation by experts. In contrast to code static analysis, penetration testing [37] is used when the code can already be executed. It works by emulation of security attacks to check for exploitable vulnerabilities. Both static analysis tools and penetration testing tools have limitations and their low effectiveness in detecting vulnerabilities has been shown in several studies [27], [38].

Our work aims to help developers focusing on security in the early stages of software coding, by collecting and analysing measurable evidences of security issues in the code.

Several approaches are used in the literature to deal with vulnerability prediction based on evidences and data collected from the source code [39], namely: software metrics, text mining, dependency graphs, and taint analysis. In this paper, we focus on **software metrics**, which are widely used as indicators of software quality (e.g., reliability and maintainability) [40]–[42]. It is worth noting that comparison between software metric based approach and other approaches are out of focus of this paper. Using software metrics for training models to predict software bugs (not necessarily security issues) is not a new topic [43]–[46]. A survey of various machine learning algorithms with software metrics for prediction of software faults is presented in [47].

Several studies in the literature show that there is some correlation between software metrics and security vulnerabilities [48], [49]. We can also find several works related to the detection of security issues using data mining, machine learning, and statistical techniques combined with software metrics [50]. However, most of these studies and works are either done over a limited number of software metrics (e.g., complexity metrics) [48], [51], [52], or use a combination of software metrics with other features [53], [54], or focus on a single security issue (e.g., buffer overflow) [55], or are limited to a specific code unit (e.g., file/class or function/method) [54] and a specific software project [11], or make a comparison between software metrics and other features [56]. To the best of our knowledge, there is no comprehensive study on software metrics and their capabilities for the detection/prediction of security vulnerabilities in the code.

In a previous work [57], we tried to address some of the limitations mentioned above by performing an analysis over a large number of software metrics obtained from different software projects, to demonstrate the possibility of using such metrics as an indicator of the existence of vulnerabilities. A heuristic search algorithm (Genetic Algorithm) combined with one classifier model (Random Forest) was used to find the most relevant subset of software metrics leading to a prediction model with the higher *accuracy*. Although the results obtained suggest that software metrics can be used to distinguish vulnerable code with a high level of accuracy, the work is quite limited, as *accuracy* is not the best criteria when dealing with imbalanced datasets. Furthermore, the work was limited to a single classification model.

In this work, we do not just aim at building one prediction model using a machine learning algorithm over software metrics as many works in the literature; we perform a comprehensive experiment to study several machine learning algorithms, several combination of software metrics, several application scenarios, several code levels, and several software projects individually and in combination, in order to find out how to achieve the best result within different circumstances for different scenarios and how to generalize the results.

The work most similar to ours is the one presented in [58]. The authors compare several state of the art machine learning techniques [54], [59], [60] regarding their ability to detect vulnerable code. There are, however, several shortcomings in that work: *i)* the study is limited to file level metrics; *ii)* it is limited to the use of the same single set of software metrics in all experiments; and *iii)* the projects included in the dataset are not considered individually in the experiments, but in combination, which limits the conclusions.

III. METHODOLOGY

Our goal is to conduct a comprehensive study to understand whether software metrics can effectively be used to distinguish the vulnerable code units from the non-vulnerable ones. The experimental process is divided in two phases, as shown in Fig. 1. The first phase, **Preliminary Analysis**, is focused on defining the configurations and settings to be used in the experiments. These configurations and settings are mainly related to the specification of the dataset (dimensions and class distribution), machine learning algorithms to be used for building the classification models, and definition of the scenarios under which the models will be evaluated. The second phase, **Experimentation and Analysis**, is focused on running the experiments based on the configurations defined in the previous phase, and analyzing the results obtained. In practice, these experiments involve building and evaluating classification models using different machine learning algorithms, different combinations of software metrics, and source code of different software projects within different

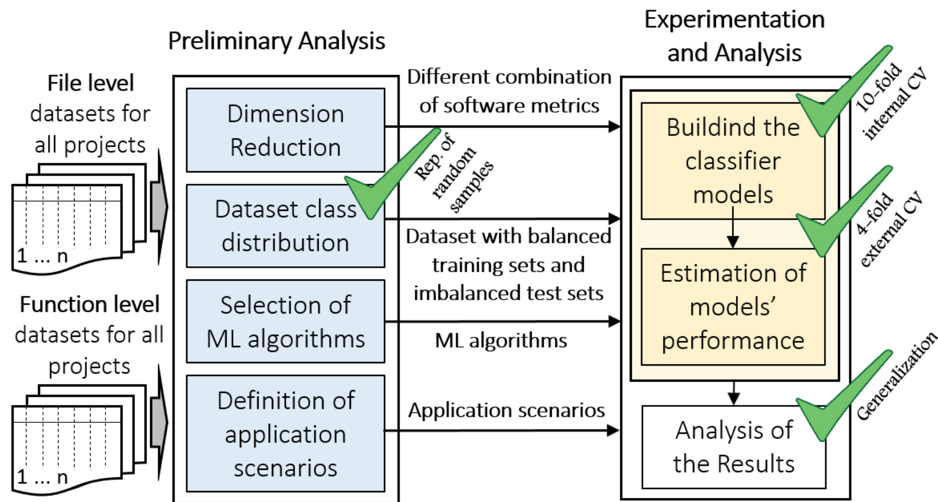


FIGURE 1. Methodology used in this work.

application scenarios. In addition to and integrated with the above principal phases, we validate the approach and methods used as well as the results obtained and demonstrate whether the results can be generalized. These **Validation and Generalization (V&G)** activities are shown in the Fig. 1 with green check marks. In short, our study considers:

i) Five representative software projects (Mozilla Firefox, Linux Kernel, Apache HTTPd, Xen and Glibc), used both individually and in combination;
ii) Five combinations of software metrics of different types (complexity, volume, coupling and cohesion) collected at different levels of code (file and function);
iii) Five widely-used machine learning algorithms (Random Forest, Extreme Boosting, Decision Tree, SVM Linear and SVM Radial), considering different configurations to achieve the best prediction results;
iV) Four application scenarios with diverse concerns regarding security (highly-critical, critical, low-critical, non-critical), which in practice are addressed by using different evaluation criteria (Recall, Informedness, F-Measure, Markedness).

The dataset used in this study, available at [61] and described in [12], contains detailed information about the whole source code, composing files, classes, and functions of several versions of five software projects implemented in C/C++: Mozilla Firefox (mozilla.org), Apache HTTPd (httpd.apache.org), Linux Kernel (kernel.org), Xen Hypervisor (xen.org), and Glibc (gnu.org/software/libc). This information was extracted by Alves *et al.* from the source code of several versions of the aforementioned projects using Understand [62], and represented through a long list of software metrics. Our analysis is performed on different architectural levels of these projects, namely file and function levels, each one having its own set of software metrics. It is worth noting that class-related metrics are not considered in this work as only one of the projects (Mozilla Firefox) is implemented in an object-oriented language, C++ in the case, and therefore contains classes.

As mentioned, we use a large set of software metrics of different types, including complexity (e.g., Cyclomatic Complexity), volume (e.g., Lines of Code), coupling (e.g., Coupling Between Objects), and cohesion (e.g., Lack of Cohesion) metrics. In practice, a total of 28 function-level metrics and 51 file-level metrics are considered (the complete list of metrics used in this work can be found in Table 3 and Table 4, and their description can be found in [62]). The dataset also includes detailed information about the known vulnerabilities, obtained by analyzing of a large number of security patches gathered from various sources (CVEDetails, Mozilla Foundation Security Advisores (MFSA), and Xen Security Advisores (XSA)). Table 1 presents a summary of the projects and their vulnerabilities. It is worth mentioning that only source files (*.c* and *.cpp* files) are considered in our analysis, so the number of functions, files and lines of code presented in Table 1 do not include the information in C header files (*.h* files) that only contain function declaration and not implementation. As shown in the table, Linux Kernel and Mozilla Firefox are the biggest projects in terms of the number of files and functions and Apache HTTPd is the smallest one. In all projects, the percentage of vulnerable code is quite low, so that we need to somehow deal with highly imbalanced dataset when building the models. It becomes worse in the case of functions. Among all, project Glibc is extremely imbalanced. More details about the dataset can be found online [61].

We selected this dataset because it includes various versions of several important and representative projects from a security point of view: they are used by many worldwide users, they were already targeted by many security attacks, and they include several versions of the same file or function (including versions with and without vulnerabilities). This is important for building more effective prediction models

TABLE 1. Summary of the dataset.

Software Projects	# Files			# Functions		
	Total	# Vulnerable	% Vulnerable	Total	# Vulnerable	% Vulnerable
Linux Kernel	383622	8712	2.27%	1910776	3021	0.16%
Mozilla Firefox	185994	3379	1.82%	1418482	2780	0.20%
Apache HTTPd	3031	94	3.10%	17046	50	0.29%
Xen	6196	278	4.49%	35430	241	0.68%
Glibc	21843	94	0.43%	23790	24	0.10%

for real cases, namely for contexts were the same project has many different versions and past knowledge should be used for preventing future vulnerabilities. Each project in the dataset is representative of a broader class of software in a particular category, in terms of functionality (e.g., the Apache HTTPd can be considered representative of HTTP servers). To the best of our knowledge, this is the most complete and extensive dataset available fitting our purposes.

A. PRELIMINARY ANALYSIS

The first phase of our exploratory study is focused on the configuration and settings of the experiments. These configurations and settings are related to: *i)* reduction of the dataset dimension; *ii)* adjustment of the dataset class distribution; *iii)* selection of machine learning algorithms; and *iv)* definition of application scenarios and selection of appropriate evaluation criteria for the scenarios.

1) DIMENSION REDUCTION

Although some software metrics may contain useful information to detect vulnerable code units, others might be irrelevant or redundant. This way, in order to build a high performance classification model out of software metrics for vulnerable code detection, it may be important to search for the most informative and discriminative metrics and to discard the redundant or irrelevant ones, which may reduce the accuracy and the computational efficiency of the classifier [63]. In this step, we aim to find out whether the process of reducing the number of features under consideration can indeed help achieving better results.

There are several strategies to deal with the issue of identifying the less-informative software metrics [64]. The basic strategy is called *exponential search*, which is the most exhaustive search technique, guaranteeing that the optimal subset of software metrics is found. Nevertheless, this strategy is not promising or not feasible in practice when the number of features (software metrics in our work) is high (for a feature set of size n, the number of iterations would be 2^n). Another strategy is *heuristic search*, which tries to guarantee the convergence to the (near) best subset of software metrics. This strategy is time consuming and its results depend on the classification model that is used as fitness function. Finally, *statistical-based filtering* can be used to find out which metrics *may* not be informative for the detection of vulnerable code units. In this work, we use this last strategy, since it is relatively fast and independent from the classification models.

Fig. 2 presents the process for dimension reduction. As shown, we conducted a detailed correlation and redundancy analysis on the software metrics at file and function levels for the five projects included in the dataset. These analyses allow identifying the least relevant or irrelevant metrics (i.e., not or lowly correlated with the class under study, which is the existence of vulnerability), and the redundant software metrics (with respect to other metrics).

To identify the irrelevant metrics, we calculate the correlation between metrics and the existence of vulnerabilities using two well-known techniques: *Pearson* [65] and *Spearman* [66] correlation coefficients. While the first evaluates the linear relationship between the software metrics and the existence of vulnerabilities, the second evaluates the monotonic relationship between them. Note that, in this work, we use both Pearson and Spearman correlation coefficient techniques to distinguish highly correlated features (i.e., when value of one feature increases then the value of other feature increases by a consistent amount) from the irrelevant ones.

Once the calculations are done, the software metrics are ranked by correlation value (from the highly correlated metrics to the least correlated ones). To select the irrelevant software metrics from this ordered list, a threshold should be defined. In this work, we consider the median as a threshold, as it is commonly used in the literature [67]. Thus, **the software metrics with both Pearson and Spearman correlation values below the median are considered as Irrelevant**.

To identify the redundant metrics, the Markov Blanket Filtering [68], [69] is used. Based on this filtering technique, let G be the current set of software metrics. If software metric (SM) SM_j has a Markov Blanket SM_i within G, it suggests that SM_j contributes with no more information beyond SM_i to the target class (i.e., existence of vulnerability in this work), and, therefore, SM_j can be safely removed from G. Based on the Approximate Markov blanket definition from [68], given two predictive software metrics SM_i and SM_j and the target class V, SM_j is redundant to SM_i, if both equations 1 and 2 are true:

$$C(SM_i, V) \geq (SM_j, V) \quad (1)$$

$$C(SM_i, SM_j) > C(SM_j, V) \quad (2)$$

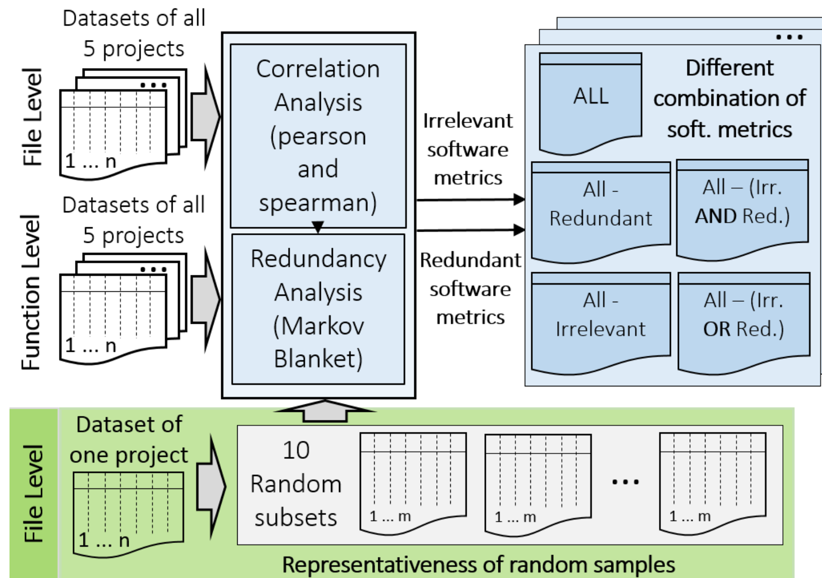


FIGURE 2. Dimension reduction process.

where, $C(SM_i, V)$ is the correlation coefficient between SM_i and the target class V, $C(SM_j, V)$ is the correlation coefficient between SM_j and the target class V, and $C(SM_i, SM_j)$ is the correlation coefficient between the two predictive software metrics SM_i and SM_j. For this analysis, we again use both Pearson and Spearman techniques to calculate the correlation coefficient. In practice, **we consider software metrics as Redundant when they are identified as so (based on the Approximate Markov blanket), using both Pearson and Spearman techniques.**

After identifying the irrelevant and redundant metrics, we generate 5 groups of software metrics to be analyzed in further experiments (the goal is to understand whether dimension reduction based on correlation and redundancy analyses can help to achieve better results):

i) **All**, which includes all software metrics present in the dataset;
ii) **All - Irrelevant** that includes all metrics minus the ones that are considered as irrelevant;
iii) **All - Redundant**, which includes all metrics minus the ones that are considered as redundant;
iv) **All - (Irrelevant AND Redundant)** that includes all metrics minus the ones that are listed as irrelevant and as redundant; and
V) **All - (Irrelevant OR Redundant)**, including all metrics minus the ones that are listed as irrelevant or as redundant.

2) CLASS DISTRIBUTION IN THE DATASET

As shown in Table 1, the dataset used in this work is quite imbalanced, as the vulnerable code units make a small fraction of the whole dataset (e.g., 2.27% in the case of Linux Kernel files). In such cases, research shows that machine learning algorithms tend to be overwhelmed by the large class and ignore the small ones [70]. On the other side, transforming a representative dataset into a balanced dataset (either by undersampling or by oversampling) may cause the loss of information about the frequency of each class and, thus, affecting the accuracy of the classification models [71]. For this reason, we performed an analysis to find out how balanced the dataset should be in order to build high performance classifiers for vulnerability detection (i.e., models with high true positive and low false positive rate). In practice, we apply one of the most effective (in terms of performance) and efficient (in term of time) strategies to deal with imbalanced data, which is to moderately undersample the majority class [72], to gradually balance the dataset (from a fully representative and imbalanced dataset to a 100% balanced dataset) and observe the impact on the performance. This allows to select a dataset with the near best class distribution that results in the near best performance compared to others.

✓V&G - Representativeness of Random Samples: In some experiments, we do not use the whole dataset but a random sample of it (as a result of undersampling, which is done to balance the training sets helping to build more effective models, as explained in Section III-A2). For this reason, it is necessary to demonstrate that the randomly chosen samples are representative of the whole dataset and, thus, do not influence the overall results. To do so, we perform a correlation and redundancy analyses over 10 different random samples, including 10000 records each, from one project, namely Firefox. This is done in order to demonstrate that the randomly selected samples follow similar statistical patterns, thus, are able to build pretty much similar predictive models (to ensure that there is no sampling bias influencing the analysis).

3) MACHINE LEARNING ALGORITHMS

Our work is focused on the idea of using machine learning algorithms for detecting vulnerable code units based on software metrics. Thus, we selected several commonly used or recommended machine learning techniques to thoroughly explore this idea. By referring to [50], [58] that survey prediction models used for detecting vulnerabilities, the ones that seem to be the most commonly used in this area are: Decision Tree [73], Random Forest [74], Support Vector Machine [75], [76], and Logistic Regression (LR) [77]. Since, in practice, LR and SVM with linear kernel usually present similar results [78], we use linear SVM in addition to radial SVM and discard LR. In addition to these, we also include the Extreme Gradient Boosted [79], as its good performance has been shown in many cases [80]. In short, the machine learning algorithms used in this study are:

- **Decision Tree (DT)**: commonly and most used supervised learning technique to support decision making. Given a dataset composed of several features and target classes, by using the Decision Tree technique, a sequence of classification rules are generated to make decisions in diverse cases. To generate these rules, it uses a tree-like model to break up a complex decision into several simpler decisions [73].
- **Random forest (RF)**: is one of the most popular ensemble learning algorithm. This algorithm consists of a combination of several DT-based classifiers, each one fitted on a random sample of a dataset, making it more accurate and robust to outliers and noise than a single DT-based classifier [74].
- **Extreme Gradient Boost (EGB)**: a specific implementation of the Gradient Boosting method that uses more accurate approximations to find the best tree models. Its main difference compared with random forest is that it builds one tree at a time. Each new tree helps to correct errors made by the previously trained tree. EGB models are becoming popular due to their effectiveness at classifying complex data [79], [81].
- **Linear Support Vector Machine (SVM)**: SVM is another widely used supervised machine learning algorithm, which is usually used for solving classification problems with two classes. Linear SVM performs classifications by finding a line that best differentiates the target classes by maximizing the margin between them [75].
- **Radial Support Vector Machine (SVM)**: a nonlinear or radial SVM applies the kernel trick to find a hyperplane (decision surface), instead of a line, to best separate two classes, when there are non-linear interactions in the data. It does a non-linear transformation on the features and converts them to a higher dimensional space to add non-linearities to the learning process [76].

All of the above algorithms are used to perform supervised machine learning. Supervised classification requires that the data is totally labeled, as is the case in our work. The algorithms are tuned to achieve the best prediction result at the cost of having longer training time. In the case of Xboost, Linear and Radial SVM, a list of values (based on literature) are given to the algorithms for each parameter to try different combinations and the best result is selected in each case. In the case of Random Forest and Decision Tree, the recommended default values from the literature are used for each parameter.

4) APPLICATION SCENARIOS AND DECISION CRITERIA

To improve the effectiveness of machine learning algorithms, it is important to adequate the evaluation criteria to the relevant application contexts. We consider four distinct scenarios where security assurance has different levels of relevance, depending on the criticality level of the applications being developed and also on the availability of resources to deal with security problems. The four scenarios analyzed were adapted from [82], where the authors define different real-world scenarios of applications to benchmark static analysis tools. We analyzed the specific characteristics of each scenario and selected an appropriate criterion associated to each one in order to evaluate the classifiers built on top of the selected software metrics. The scenarios and associated criteria are:

- **Highly-Critical**: this scenario represents highly business or safety critical systems with demanding security requirements (e.g., e-banking and e-health), in which the detection and elimination of security vulnerabilities is of high priority (because a successful security attack may cause serious damages to the system, to business, or to people's life). Thus, the classifier models should be able to detect the highest number of vulnerable code units, even if some false positives are reported. For this scenario, we choose **Recall** as criterion to evaluate the classifiers, as it measures the ratio of vulnerable code units that are correctly classified independently from false positives.
- **Critical**: this scenario represents not highly but still critical systems (e.g., e-commerce web applications and large scale social networks) in which an exploited vulnerability usually reflects sensitive data breaches or considerable financial losses. In such scenario, classifiers should detect the highest number of vulnerabilities while avoiding reporting too many false positives as the resources available to fix and remove vulnerabilities need to be used appropriately. For this reason, we chose **Bookmaker Informedness** as criterion, as it still gives a high importance to true positive rate while moderately penalizing classification models with high false positive rates.
- **Low-Critical**: this scenario includes systems that are less critical and less exposed to attacks. Projects developing these systems usually have limited budget to be allocated for finding and fixing vulnerabilities. Thus, both detecting and eliminating the highest number of vulnerabilities and spending less resources for analysing false positives have equal priority. In this scenario,

TABLE 2. Summary of the application scenarios and their corresponding criteria [83].

Scenario	Criterion	Formula	Definition
Highly-Critical	Recall	$\frac{TP}{P} = \frac{TP}{TP+FN}$	Represents the ratio of vulnerable code units that are correctly classified as vulnerable.
Critical	Bookmaker Informedness	$\frac{TP}{P} - \frac{FP}{N} = \frac{TP}{TP+FN} - \frac{FP}{TN+FP}$	Combines TP and FP rates but still gives a high importance to the number of vulnerable code units that are correctly classified and moderately penalizes classification models with high number of FP.
Low-Critical	F-Measure	$2 * \frac{precision * recall}{precision + recal} = \frac{2 * TP}{2 * TP + FN + FP}$	Represents the harmonic mean of Recall and Precision, thus evenly combines TP and FP rates.
Non-Critical	Markedness	Precision + Inverse Precision − 1 $= \frac{TP}{TP+FP} + \frac{TN}{FN+TN} - 1$	Quantifies how consistently the outcome has the classifier as a marker. It does consider both true and false alarms, but in practice, it rewards the low false positive rate.

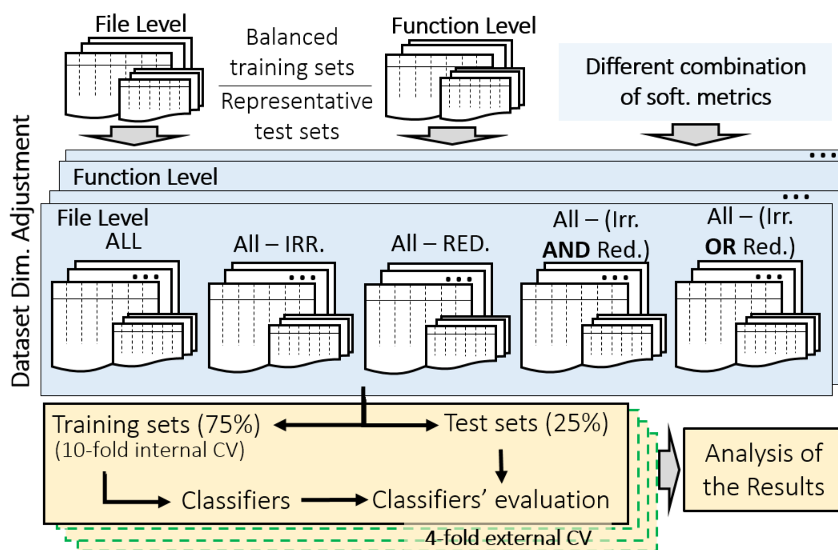


FIGURE 3. Process of experimentation and analysis.

F-Measure that evenly combines precision and recall, is an appropriate criterion.

- **Non-Critical**: this scenario includes non-critical systems from a security perspective (i.e., systems that are not usually exposed to attackers). Thus, we are more concerned with the number of false alarms due to tight budget and resource restrictions, although we still want to detect vulnerable code and eliminate vulnerabilities. **Markedness** is an appropriate criterion in this context, as it rewards low false alarms and at the same time does not ignore true positives.

More details about the selected criteria are presented in Table 2. In the formulas, True Positive (TP) represents the number of vulnerable code that are correctly classified, True Negative (TN) represents the number of non-vulnerable code that are correctly classified, False Positive (FP) represents the number of non-vulnerable code units that are misclassified as vulnerable and False Negative (FN) represents the number of vulnerable code that are misclassified as non-vulnerable.

B. EXPERIMENTATION AND ANALYSIS

The second phase of the study consists of running the **experiments**. As shown in Fig. 3, the data (balanced training sets and representative test sets belonging to all projects at both file and function levels) are prepared according to the configurations determined in the previous phase and then passed to the selected machine learning algorithms. The classification models are built over the dataset of the five different projects at file and function levels by considering the several combinations of software metrics and the different application scenarios. It is worth noting that the machine learning algorithms are trained using balanced training sets and tested using a representative test set in order to build more accurate vulnerable code detectors and have more realistic performance estimations. Internal and external cross-validation (CV) is performed in all cases, as discussed next.

✓**V&G - Internal Cross Validation:** In order to avoid any overfitting that might be caused by unrepresentative training sets, internal cross validation is necessary. Cross-validation

is a statistical resampling technique used to estimate the performance of machine learning models [84]. Using this technique, data is split into k subsets or folds of equal size. Each time, one fold is used as test set and the remaining k-1 folds are used to train and fit the model. In this work, we use an **internal 10 fold cross validation** for building the models, helping to achieve a fair estimation of the performance for each individual model.

✓**V&G - External Cross Validation:** Internal cross validation might not be enough to ensure fair comparison between distinct models due to the fact that the initial training and test sets might not be representative of the whole dataset. For this reason, in this work, we use an **external 4-fold cross validation** to validate the classification models built. In practice, we divide the whole dataset into 4 folds. Each machine learning algorithm is executed four times; each time, it uses one fold for testing and 3 folds for training (which internally uses a 10 folds cross validation). The final performance estimation of each classification model is an average of the four estimations.

✓**V&G - Generalization Assessment:** We perform two sets of tests to understand to which extent we can generalize the obtained results. The first set of tests is focused on inter-project cross assessment. In these tests, the machine learning algorithms are trained using the dataset of one project (e.g., Linux Kernel) and are tested using the dataset of the other projects. This helps to understand how machine learning algorithms perform in new situation. In the second set of tests, the machine learning algorithms are trained using a combined dataset including all projects and tested using the dataset of each project individually. This helps to understand whether it is helpful to combine all existing information from source code of different software projects to achieve a better result.

IV. PRELIMINARY ANALYSIS PHASE

We used the R Project [85] and several R libraries, including CARET [86], RandomForest [87], e1071 [88], and dplyr [89] to perform the experiments. All the experiments were executed on virtual machines with Ubuntu 16.04.6 LTS, a 2.0 GHz Intel Xeon E312xx (Sandy Bridge) processor, 8GB RAM and 16MB cache. In this section, we present and analyze the results obtained during the preliminary analysis phase.

A. DIMENSION REDUCTION

Correlation and redundancy analysis were performed for all projects at both file and function levels. Tables 3 and 4 present the results obtained for both levels. Although the list of irrelevant or redundant metrics identified are not the same in all projects, we can see a high level of similarity between them. For instance, as shown in Table 3, from the 27 file-level metrics (out of a total of 51 metrics) that are considered as *irrelevant* in all five projects, 25 appear at least in 3 projects (e.g., AvgCyclomatic, AltAvgLineBlank, AvgCyclomaticModified, AvgCyclomaticStrict). Similarly, from the 38 file-level metrics considered as *redundant* in all projects, 26 appear at least in 3 projects (e.g., AvgCyclomatic, CountLineBlank, CountLineCodeExe, CountSemicolon). Despite these similarities, in order to be more precise, we run our experiments over the 5 groups of software metrics (i.e., *All*, *All - Irrelevant*, *All - Redundant*, *All - [Irrelevant AND Redundant]*, and *All - [Irrelevant OR Redundant]*) that were created separately for each individual project at both file and function levels, which are used as input features for the machine learning algorithms.

B. CLASS DISTRIBUTION IN THE DATASET

One important factor to build high performance classifiers (i.e., classifiers with high true positive and low false positive rate) is related to the distribution of the classes (i.e., vulnerable and non-vulnerable code units in this work) in the dataset. As explained before, we used undersampling to gradually balance the dataset (from fully imbalanced dataset to 100% balanced) and observed the impact on performance. Table 5 presents the characteristics of those resampled datasets. This study is performed at file level for the Linux Kernel project by using various machine learning algorithms. Linux Kernel was chosen for this analysis due to the fact that it has a higher number of reported vulnerabilities than other projects, so a low number of vulnerable records would not be a threat to the validity of the results.

In all experiments, 75% of the resampled dataset was used to train the machine learning algorithms and 25% of it (disjoint from the training sets) was used to test them (**TS1**). In addition, to guarantee a fair and representative evaluation of the classification models, we (randomly) created an additional test set composed of 25% of the whole dataset (**TS2**), which is fully imbalanced and is ensured to be disjoint from the training sets. By doing this, we aim to understand how the estimation made by a balanced test set differs from the estimation made by an imbalanced, but representative test set.

Fig. 4 (x-axis: % of vulnerable records in the dataset (from 2.27% to 50%), y-axis: true positive rate (left) and false positive rate (right)) shows how performance, in terms of true positive rate and false positive rate estimated using resampled test set (TS1), changes when the training set becomes more balanced. For all machine learning algorithms, we observe that the true positive rate increases (e.g., from 0.54 to 0.92 in the case of Random Forest and from 0.08 to 0.73 in the case of Decision Tree). This means that more vulnerable code units are detected and less vulnerable code units are misclassified as non-vulnerable. Thus, for highly critical systems where one wants to detect as many vulnerabilities as possible (regardless of the false alarms), it is quite effective to balance the dataset when the number of vulnerable records is lower than the number of non-vulnerable ones.

Another observation is that the false positive rate increases for all algorithms (e.g., from 0.003 to 0.08 in the case of Random Forest and from 0.0007 to 0.31 in the case of Decision Tree), which means that a higher number of non-vulnerable

TABLE 3. Irrelevant and redundant file-level software metrics (a) and their frequency in 10 random samples of Mozilla Firefox (b).

(a)							(b)	
Irrelevant (I) and Redundant (R) File level metrics							# of samples	
#	Software Metrics	MOZILLA	KERNEL	XEN	APACHE	GLIBC	Irr.	Red.
1	AvgCyclomatic	I / R	I / R	I / R	I / R	I / R	9	10
2	AltAvgLineBlank	I / R	I / R	I / R	I	R	9	7
3	AvgCyclomaticModified	I / R	I	I	I / R	I / R	8	7
4	AvgCyclomaticStrict	I	I / R	I	I / R	I / R	8	1
5	AvgLine	I	I / R	R	I	I / R	9	0
6	FanOut	I / R	I	I / R	I / R	I	10	1
7	FanIn	I	I	I	I	I / R	10	0
8	SumMaxNesting	I	I	I	I	I / R	10	0
9	MaxMaxNesting	I	I	I	I	R	10	0
10	HK	I	I	I	I	I	9	0
11	LCOM	I	I	I	I	I	9	0
12	MaxFanIn	I	I	I	I	I	10	0
13	CountPath	I	I	I	I	I	10	0
14	CBO	I	I	I	I	I	10	0
15	CountLineBlank	R	R	R	R	R	0	9
16	CountLineCodeExe	R	R	R	R	R	0	10
17	CountSemicolon	R	R	R	R	R	0	10
18	CountStmt	R	R	R	R	R	0	10
19	CountStmtExe	R	R	R	R	R	0	10
20	MaxCyclomatic	R	R	R	R	R	1	9
21	MaxCyclomaticModified	R	R	R	R	R	1	6
22	AltAvgLineCode	I / R	I / R		I / R	I / R	10	9
23	AvgLineCode	I / R		R	I / R	I / R	10	8
24	AvgLineBlank	I	I	I	I / R		9	1
25	CountLineCode		R	R	R	R	0	7
26	SumCyclomatic	R	R	R		R	0	9
27	SumCyclomaticModified	R	R	R	R	R	0	8
28	AltCountLineCode	R	R		R	R	0	9
29	AltCountLineComment	R	R	R		R	0	9
30	SumCyclomaticStrict	R		R	R	R	0	8
31	RatioCommentToCode	I	I		I	I	10	0
32	CountStmtEmpty	I	I	I		I	7	0
33	AvgFanIn	I	I	I	I		10	0
34	AltAvgLineComment	I / R	I / R	I			9	10
35	AvgEssential	I			I	I / R	8	0
36	AvgLineComment	I	I	I / R			8	0
37	AltCountLineBlank		R	R	R		0	6
38	CountDeclFunction		R	R		R	1	0
39	CountLine	R		R		R	0	8
40	CountLineCodeDecl	R		R	R		0	10
41	CountStmtDecl		R	R	R		0	0
42	CountLineInactive		I	I	I		2	0
43	MaxFanOut	I		I	I		10	0
44	AvgMaxNesting	I				I / R	7	0
45	MaxCyclomaticStrict			R		R	0	2
46	AvgFanOut					R	8	0
47	CountLineComment					R	0	2
48	SumEssential				R		0	6
49	CountLinePreprocessor				I		0	0
50	MaxEssential						0	0
51	MaxNesting						0	0

code units are misclassified as vulnerable. Thus, for scenarios in which there are limited resources for fixing or removing vulnerabilities, undersampling the non-vulnerable class to balance the dataset does not seem to be a good approach. Similar results are obtained for true positive rate and false positive rate, using the imbalanced test set (TS2). This way, since we are more concerned about detecting vulnerable code units and aim to improve the tools and techniques in this regard, **we have decided to use the totally balanced (50% vulnerable code units) datasets for training the machine learning algorithms.**

We also conducted a more detailed comparison between the classifiers using balanced and imbalanced *test sets*. For this comparison, we used all machine learning algorithms, trained using a totally balanced training set and tested using both balanced (TS1) and imbalanced (TS2) test sets, and evaluated the classification models by using the four criteria representing the four scenarios under study. As shown

TABLE 4. Irrelevant and redundant function-level software metrics.

Irrelevant (I) and Redundant (R) Function level metrics						
#	Software Metrics	MOZILLA	KERNEL	XEN	APACHE	GLIBC
1	MinEssentialKnots	I / R	I / R	I / R	I / R	R
2	MaxEssentialKnots	I / R	I / R	I	I / R	I / R
3	CyclomaticStrict	R	R	R	I / R	R
4	AltCountLineBlank	R	R	I / R	R	R
5	CountStmtExe	R	R	R	R	R
6	Cyclomatic	R	R	R	R	R
7	CountLineCodeExe	R	R	R	R	R
8	CountLineInactive	I	I	I	I	I
9	CountLinePreprocessor	I	I	I	I	I
10	CountStmtEmpty	I	I	I	I	I
11	AltCountLineCode	R	R		R	R
12	CyclomaticModified	R	R	R	R	
13	CountSemicolon	R		R	R	R
14	CountStmt	R	R	R		R
15	AltCountLineComment	R	R	I	I	R
16	RatioCommentToCode		I	I	I	I
17	CountLine	R			R	R
18	CountLineCode	R		R	R	
19	Knots	I	I			I
20	Essential	I		I / R	I	
21	CountLineBlank		R	I / R		
22	CountLineCodeDecl				I	I / R
23	CountLineComment			I / R	I	
24	CountStmtDecl			I		R
25	CountInput					I
26	MaxNesting					I
27	CountOutput					
28	CountPath					

TABLE 5. Resampled datasets (Linux Kernel files).

	# Vulnerable Files	# Non-vulnerable Files	Total	Training Set	Test Set 1 (TS1)	Test Set 2 (TS2)
Imbalanced	8712 (2.27%)	374910 (97.73%)	383622	287717	95905	95905
↑	8712 (10%)	78408 (90%)	87120	65340	21780	95905
	8712 (20%)	34848 (80%)	43560	32670	10890	95905
	8712 (30%)	20328 (70%)	29040	21780	7260	95905
↓	8712 (40%)	13068 (60%)	21780	16335	5445	95905
Balanced	8712 (50%)	8712 (50%)	17424	13068	4356	95905

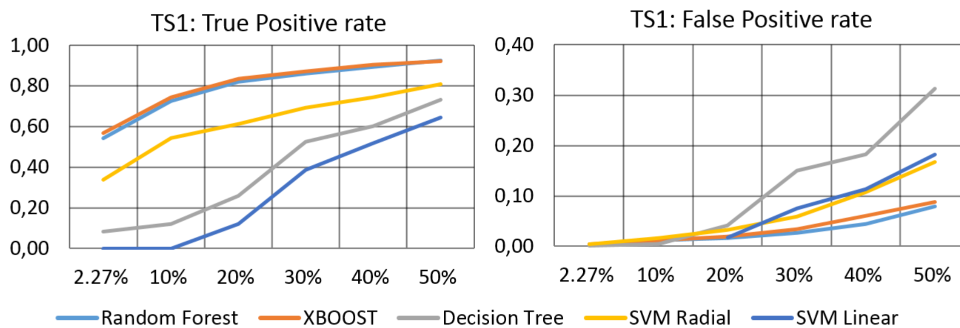


FIGURE 4. Impact of undersampling on performance.

in Fig. 5, the Recall and Informedness obtained using TS1 are in par with the results obtained using TS2. This means that using either a balanced or an imbalanced test set does not influence the classification results when highly-critical and

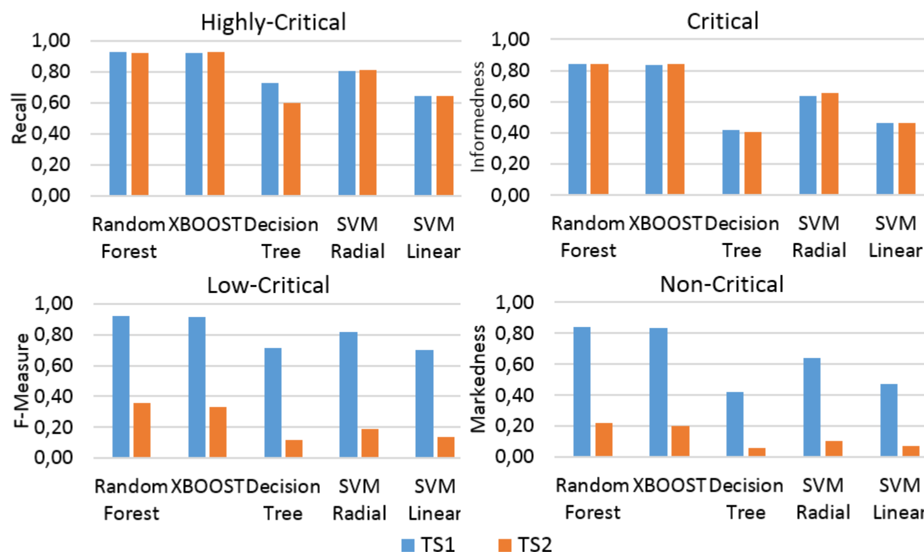


FIGURE 5. Balanced versus imbalanced representative test sets.

critical scenarios are the target of the analysis. But we observe lower values for F-measure and Markedness in TS2 across all machine learning algorithms. This is caused by the high number of false positives compared to true positives, which comes naturally from the TS2 that has a much higher percentage of non-vulnerable records. Thus, **we have decided to use imbalanced but representative test sets in the following experiments, in order to have realistic performance estimation for all scenarios.**

It is worth noting that, since this analysis is done only at file level, our decision regarding how to train and test the models may not perfectly fit in function-based experiments, but we believe that the implications are negligible, due to the fact that the nature and context of the problem is quite similar.

In order to demonstrate that the samples generated for training or testing are representative, we performed the *V&G - Representativeness of Random Samples* validation analysis. To do so, we repeated the correlation and redundancy analyses (presented in Section IV-A) ten times over 10 random samples (with 10000 records each) of file level data from the Mozilla Firefox project. Results show that in all cases, 30 software metrics (out of a total of 51 file level metrics) are identified as irrelevant and 28 software metrics are identified as redundant. In addition to that, there is a large group of metrics that appear repeatedly across different sample sets as irrelevant and redundant. For example, as shown in the last two columns of Table 3, 26 out of 30 irrelevant metrics are identified in at least 7 samples (e.g., FanOut in 10 samples, AvgCyclomatic in 9 samples). We obtained similar results regarding redundant metrics: out of a total of 28 redundant metrics, 23 are identified as such in at least 6 samples (e.g., AvgCyclomatic in 10 samples, AltAvgLineBlank in 7 samples). These results show that the random samples have quite similar characteristics and patterns in terms of correlation between the software metrics, and between the software metrics and the existence of vulnerabilities, which are important factors in building predictive models out of software metrics. One of these samples is randomly chosen for further experiments and analysis to re-ensure the avoidance of any sampling bias that may exist.

V. EXPERIMENTATION AND ANALYSIS PHASE

In this section, we present and analyze the results obtained during the experimentation and analysis phase, including the performance of the machine learning algorithms, importance of software metrics, and generalization of the approach.

A. PERFORMANCE OF THE MACHINE LEARNING ALGORITHMS

We first focus on the results obtained for each project individually and then make a comparison. Fig. 6 and Fig. 7 present the results obtained respectively for file and function level software metrics of the Linux Kernel project. Both figures include the results obtained by all machine learning algorithms for different scenarios over five combinations of software metrics. It is worth reminding that all 5 software projects are analyzed by using four criteria representing four different scenarios. In fact, the assumptions regarding the criticality level of the projects are made based on scenarios.

File level results show that the best performance is always achieved by Random Forest and Xboost algorithms. As expected from the non-linear nature of the dataset, radial SVM always achieve a better performance than linear SVM, which is almost in par with Decision Tree. Among different combinations of software metrics, the combination from which the irrelevant metrics are eliminated *slightly* shows a better result than other combinations in most cases. In Fig. 6, we can also see that the combination in which the redundant

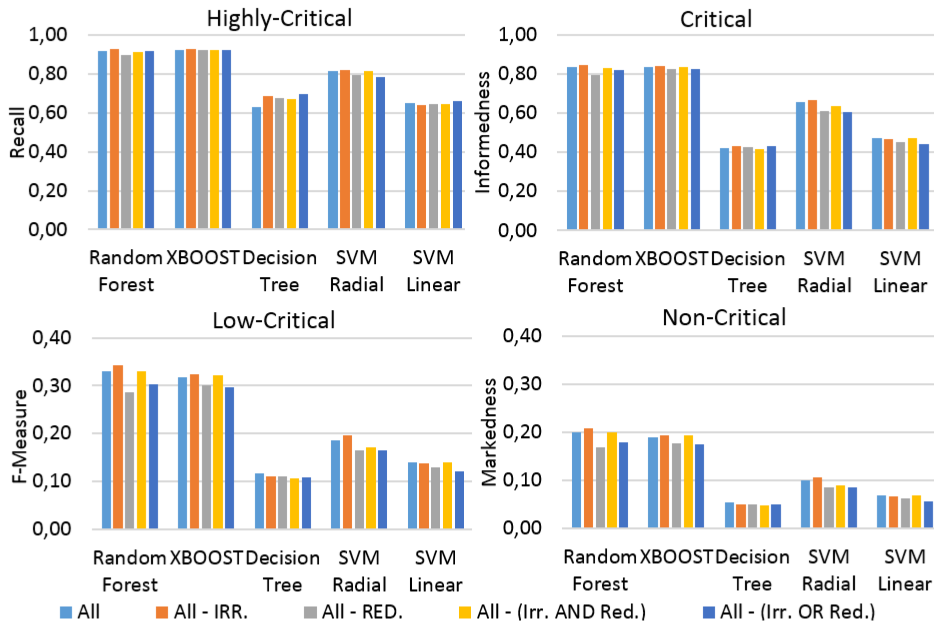


FIGURE 6. File level results for Linux Kernel project.

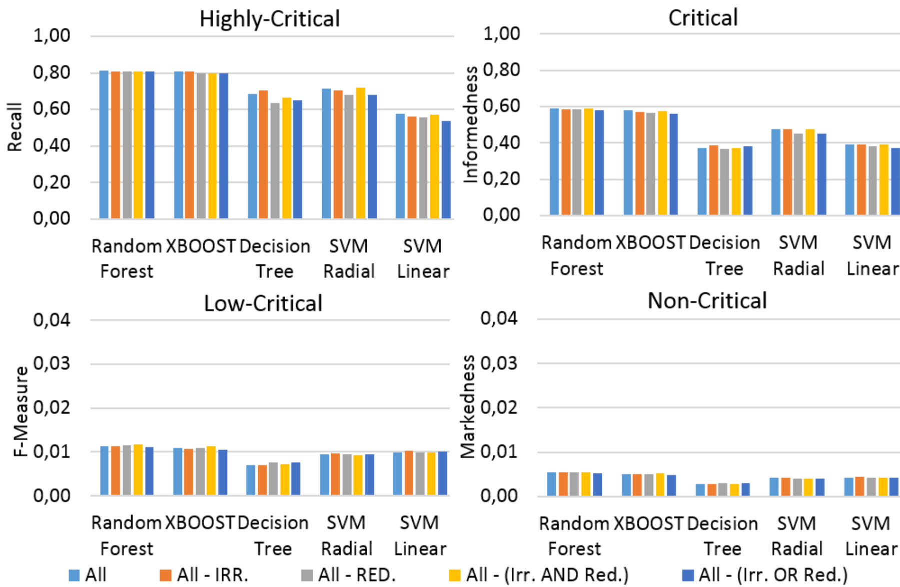


FIGURE 7. Function level results for Linux Kernel project.

metrics are eliminated shows (slightly) worse results than the combination with all metrics. In contrast, function level results show no significant difference between these combinations. This happens because the function-level dataset is not considered as a high-dimensional dataset (it only has 28 features), and in such cases it is hard to achieve a better result with dimension reduction. However, this is not always the same for other projects (see Fig. 8 and Fig. 9).

After analysing the results of all projects and all algorithms, we can state that, **dimension reduction, does not always help to achieve a better performance**. In fact, dimension reduction has to be done carefully and several techniques should be tried depending on the classification model in use and the characteristics of the dataset in order to achieve a better performance.

Regarding the effectiveness of using software metrics and machine learning algorithms to detect vulnerable code, we can conclude that, although the machine learning algorithms could achieve a reasonable performance in terms of Recall and Informedness (highly critical and critical scenarios), the results for F-measure and Markedness (low-critical and non-critical scenarios), which are highly dependent on

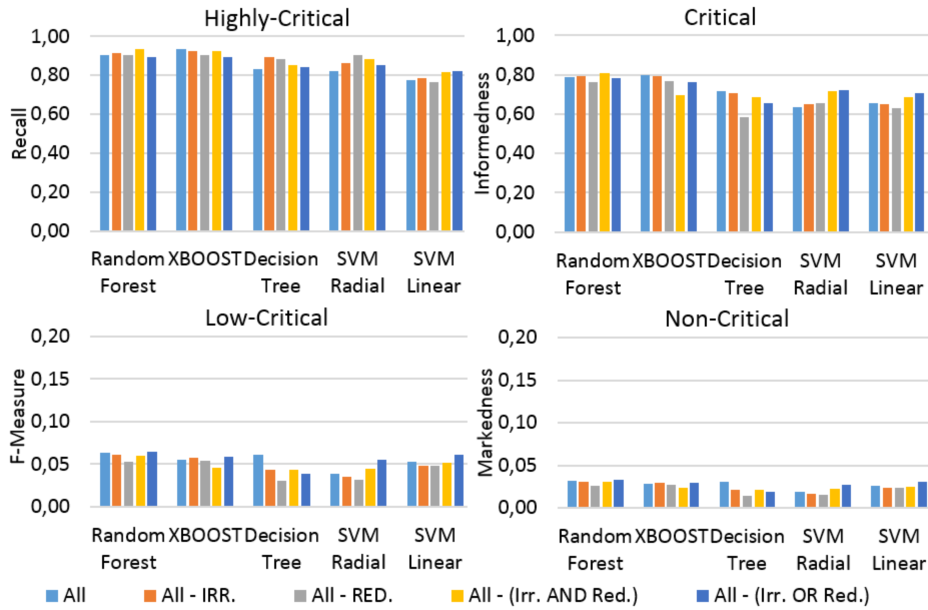


FIGURE 8. File level results for Glibc project.

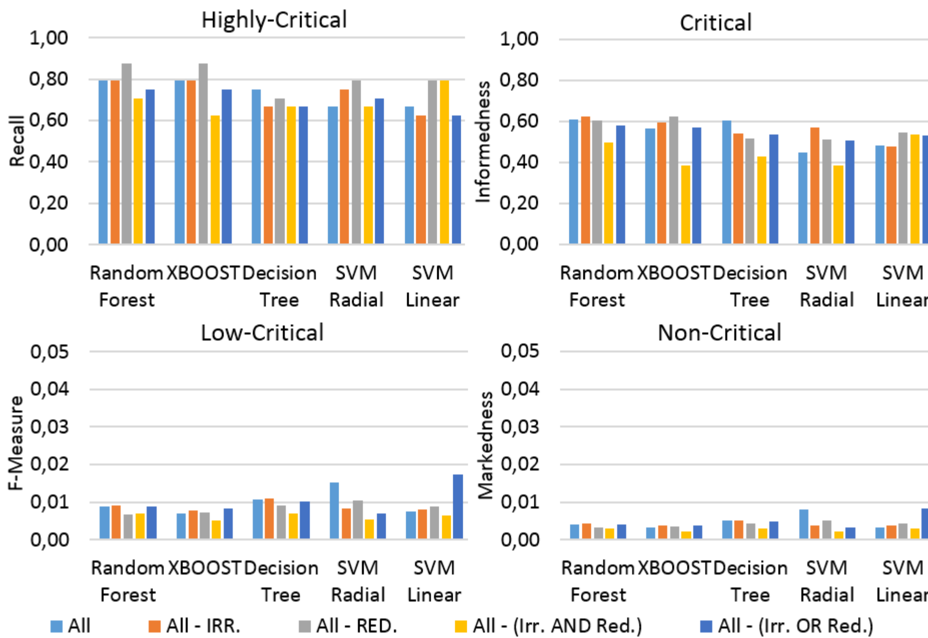


FIGURE 9. Function level results for Glibc project.

the number of false positives compared to true positives (refer to Table 2), are not convincing at all. Despite having high true positive (TPR = TP/P) and low false positive rates (FPR = FP/N), having a very imbalanced test set leads to a high number of false positive cases when compared to the number of true positive cases.

Similar observations can be pointed for the function level results presented in Fig. 7, with the difference that the performance of the algorithms using file level metrics is usually higher than when using function level metrics. Also, the difference between machine learning algorithms is more visible in file level results. For example, we cannot see any difference between the algorithms in terms of F-Measure and Markedness in Fig. 7. These are happening due to the fact that the function-level data is even more imbalanced than the file-level data (refer to Table 1).

To make a comparison between different projects, we present in Fig. 10 the results obtained for all projects over

the data sets with all file level metrics. In general, the results for the different projects are quite different mainly due to the fact that the characteristics of the datasets (i.e., size and distribution of classes) are different for each project. In most cases, the best performance is achieved for the Linux Kernel dataset, which is the biggest project and has more vulnerable code units. This means that the machine learning algorithms had more evidences and more balanced information to avoid overfitting and learn (of course not equally) about both classes involved in the dataset. We also have high Recall and Informedness for Glibc, but, by looking to the very low F-measure and Markedness values, we can conclude that the high true positive rate in this case is achieved thanks to highly overfitted models.

Interestingly, the results achieved by both ensemble algorithms, Random Forest and Xboost, are quite similar in the case of all projects, for both file and function level metrics (see Fig. 10 and Fig. 11). Random Forest and Xboost are both tree-based algorithms. In both cases, the performance of the model depends on two distinct sources of error: bias and variance. Gradient boosting models deal with these sources of error by boosting for many rounds at a low learning rate. In contrast, Random Forest models deal with them via the number of trees and tree depth. Achieving very similar results by these algorithms in almost all cases may imply that both models were able to achieve their best model with our dataset and no bias or variant could be reduced by neither methods due to the limitations that exist in the dataset (e.g., being imbalanced with imperfect labeling).

We also observe similar patterns between Linear SVM and DT. However, showing a comparable performance does not imply that the code is classified or misclassified similarly by these classifiers. For this reason, we decided to analyse their behaviour in more detail. Fig. 12 presents Venn diagrams showing all possible intersections between the subset of vulnerable code units (respectively, files and functions of the Linux kernel project) that are classified as non-vulnerable by the different machine learning algorithms. The diagram shows that 76 vulnerable files and 102 vulnerable functions are misclassified by all classifiers. Interestingly, 136 out of 146 vulnerable files and 137 out of 163 vulnerable functions that are classified as non-vulnerable by XBoost, are also misclassified by Random Forest. This led us to analyze the characteristics of the vulnerable/non-vulnerable files and functions that are misclassified by all classifiers to find the missing information that the machine learning algorithms could use to improve their performance.

In our analysis we observed that most of the vulnerable code units that are classified as non-vulnerable are small and simple in terms of structure. In contrast, most of the files and functions that are incorrectly classified as vulnerable are huge or complex. An example of a misclassified vulnerable file from Linux Kernel source code is presented below.

```c
1  /*
2   * File Path: fs/ramfs/file-mmu.c
3   *
4   * Software Metrics values:
5   * CountLineCode: 15
6   * SumCyclomatic: 0
7   * SumCyclomaticMod: 0
8   * SumCyclomaticStrict: 0
9   * SumEssential: 0
10  * CountPath: 0
11  * FanIn: 0
12  * FanOut: 0
13  */
14
15 #include <linux/fs.h>
16 #include <linux/mm.h>
17 #include <linux/ramfs.h>
18
19 #include "internal.h"
20
21 const struct file_operations ramfs_file_operations
      = {
22   .read    = new_sync_read,
23   .read_iter  = generic_file_read_iter,
24   .write     = new_sync_write,
25   .write_iter = generic_file_write_iter,
26   .mmap    = generic_file_mmap,
27   .fsync     = noop_fsync,
28   .splice_read  = generic_file_splice_read,
29   .splice_write = generic_file_splice_write,
30   .llseek    = generic_file_llseek,
31 };
32
33 const struct inode_operations
     ramfs_file_inode_operations = {
34   .setattr  = simple_setattr,
35   .getattr  = simple_getattr,
36 };
```

We added the first 13 lines just to provide some information about the file. File path is added in line 2 and the values of several representative software metrics are added in lines 4 to 12. The values of the metrics show how **simple** the file is. Indeed, it is **impossible** to indicate this file as vulnerable file by using software metrics, but we are aware of one exploitable vulnerability that has been reported for this file (i.e., *CWE-264 - Permissions, Privileges, and Access Controls*). The vulnerability consists of a **Wrong Assignment Value** (according to the ODC classification [90]) in line 29, which allows local users to cause a denial of service (system crash). To fix this vulnerability the line should be simply replaced by *.splice_write = iter_file_splice_write,*. Given this example, the analysis of the misclassified vulnerable but simple files and functions, may allow finding other evidences that help to improve the performance of vulnerability detection tools. This will be explored in future work.

Fig. 13 and Fig. 14 present the average value of several software metrics respectively for misclassified files and functions. As we can see, there is a huge difference between these two groups of misclassified code units (i.e., false positives and false negatives). Note that the standard deviation is high too, which means that the average variation around the mean is quite large. Regarding the false positive cases, as mentioned before, the source of information regarding the vulnerabilities is limited to security reports. Consequently, the functions

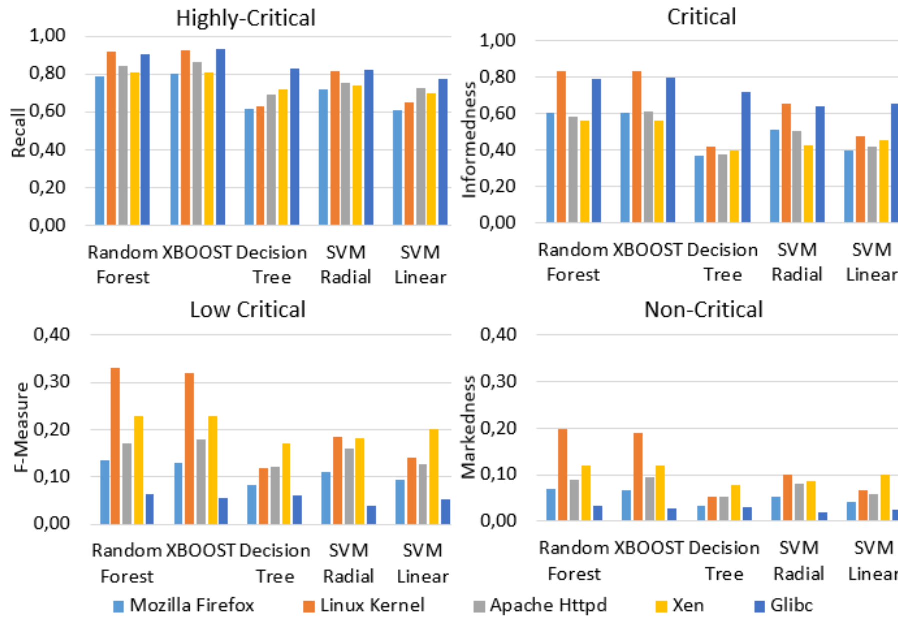


FIGURE 10. File-level results for all projects over all software metrics.

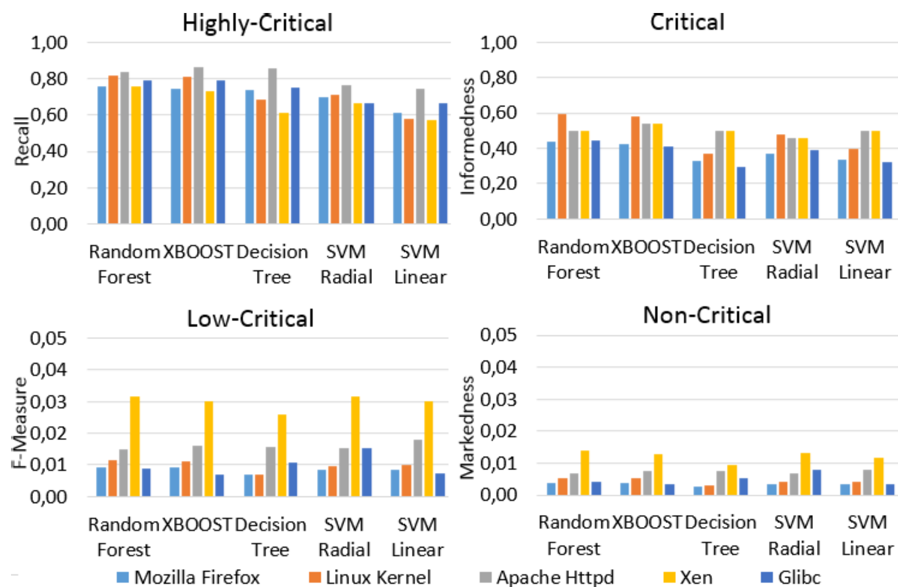


FIGURE 11. Function-level results for all projects over all software metrics.

and files without reported vulnerabilitis are not necessarily flawless. For this reason, given the above results regarding misclassified non-vulnerable files and functions, it is quite probable that some of the considered false positives are indeed vulnerable (although no vulnerability has yet been disclosed). A few lines of an example of a missclassied non-vulnerable file (i.e., no attack related to this file is reported so far) from Linux Kernel source code is presented below (the whole file is not presented due to the lack of space).

As the values of software metrics show (lines 4 to 12), the file is quite **complex**. Using our models, this file is classified as vulnerable. To find out whether this file is vulnerable or a real false alarm, we applied several static code analyser tools including FlawFinder, CppCheck, and Rats [91] over the file. These tools found several issues in the file, of which two vulnerabilities were confirmed by a security expert. One of these vulnerabilities is found in line 18 of the code below (i.e., *CWE-134 - Use of Externally-Controlled Format String*) where *sprintf* operation is used without checking the input

value (**Missing Checking Input Value** bug according to ODC classification).

```
/*
 * File Path: drivers/pcmcia/ds.c
 *
 * Software Metrics values:
 * CountLineCode: 772
 * SumCyclomatic: 208
 * SumCyclomaticMod: 206
 * SumCyclomaticStrict: 226
 * SumEssential: 115
 * CountPath: 2122
 * FanIn: 399
 * FanOut: 159
 */

static ssize_t field##_show (struct device *dev,
    struct device_attribute *attr, char *buf)    \
{                    \
  struct pcmcia_device *p_dev = to_pcmcia_dev(dev)
    ;   \
  return p_dev->test ? sprintf (buf, format, p_dev
    ->field) : -ENODEV; \
}
```

B. IMPORTANCE OF SOFTWARE METRICS

In all the experiments above, we collected information regarding the importance of the software metrics calculated by each machine learning algorithm for the five projects. In general, the rankings given by the algorithms are different from each other, as the algorithms build the models differently and the datasets of the projects have different characteristics (in terms of size, distribution of classes, and structure of the code).

To better understanding the results, we compared the ranking of the software metrics given by the two classifiers that performed higher (Xboost and Random Forest) in the two projects with the best results (Linux Kernel and Mozilla Firefox). We compared the most important (importance of software metrics refers to the score assigned to them by each classifier based on how useful they are at predicting a vulnerable code) file and function level software metrics (one-third of software metrics with highest score - 10 out of 28 function-level metrics and 17 out of 51 file-level metrics) given by each algorithm for the two projects. Table 6 presents the software metrics that appear in the list of the most important (i.e., discriminative) metrics in both Linux Kernel and Mozilla Firefox projects for each classifier at both file and function levels. As shown, when Xboost is used to build the model over the Linux Kernel and Mozilla Firefox data, 9 out of 10 most important function level metrics are the same, although with different scores and rankings. Similarly, when Random Forest is used to build the model over the Linux Kernel and Mozilla Firefox data, 6 out of 10 most important function level metrics are the same. Regarding the file-level metrics with Xboost and Random Forest, respectively, 11 and 9 out of 17 most important metrics are repeated in both projects.

Despite these *spontaneous similarities* that we can find in some cases, according to the ranking results (some of which presented above), and according to the results presented in the previous section regarding the different subsets of software metrics, it seems that the correlation between software metrics and also their correlation with security issues in the code is sufficiently complex to be identified by our simple correlation and redundancy analysis. Even the machine learning algorithms did not rank them equally (or even with a high level of similarity). Thus, giving privilege to a group of metrics for building vulnerability prediction models does not seem to be a promising idea.

C. GENERALIZATION ASSESSMENT

To understand to which extent we can generalize the results and how the machine learning algorithms perform in new (previously unseen) situations, we first performed a inter-project cross assessment, where data of a specific project are used for training and data of the other projects are used for testing. At the file level and using data of Linux Kernel as training set, we observe that the performance decreases in all projects, except in the case of the Linux Kernel itself, whose data is used for training the machine learning algorithms (see Fig. 15). An interesting observation is that Linear SVM and DT seem to make better classifications than other machine learning algorithms when the test set is completely unknown to the classifiers. This means that these machine learning algorithms build more generalizable models than other algorithms, thus being more suitable for unseen code. This is simply because, they build simpler models, which is more appropriate when the data is more non-parametric in nature (i.e., when we cannot make assumptions about the distribution of data).

At function level and using data of Linux Kernel as training set (see Fig. 16), all classifiers seem to perform similarly. Interestingly, for Low Critical and Non-Critical scenarios, Xen achieves a better result than the other project. This happens due to the fact that this small project has a more balanced test set compared to other projects (Refer to Table 1). The same model with more balanced test set, gives less false positive alarms compared to the number of true positive cases, which leads to achieve higher F-Measure and Markedness. The same observations are seen when data of Mozilla Firefox is used as training set in both file and function levels, but in other cases, when the data of the small projects are used for training, we observed that the performance of the classifiers is way lower and all classifiers perform similarly in both file and function level. This happens because the training set is small and there is not enough variation in training set.

The results of the experiments in which the machine learning algorithms are run over the combined dataset, are presented in Fig. 17 and Fig. 18 for files and functions, respectively. We can observe that the performance of the classifiers is *slightly* degraded when we use a dataset composed of all 5 projects for building the classification models. This potentially means that classifiers are able to find similar characteristics and patterns in the code of five different projects, thus achieving a reasonable performance level. The results are similar for function level metrics. This is a promising observation as it may mean that we can build

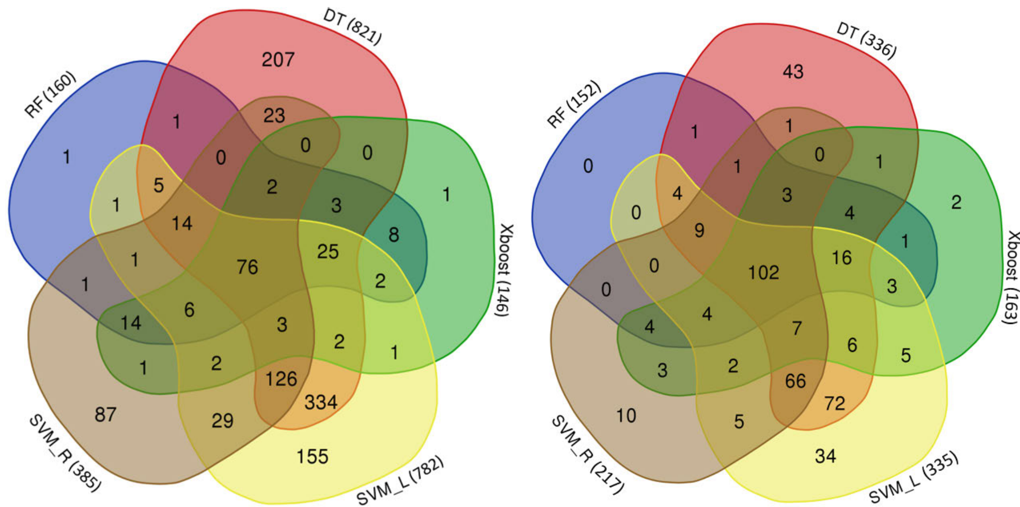


FIGURE 12. Venn diagram of misclassified vulnerable files (a) and misclassified vulnerable functions (b) of Linux Kernel.

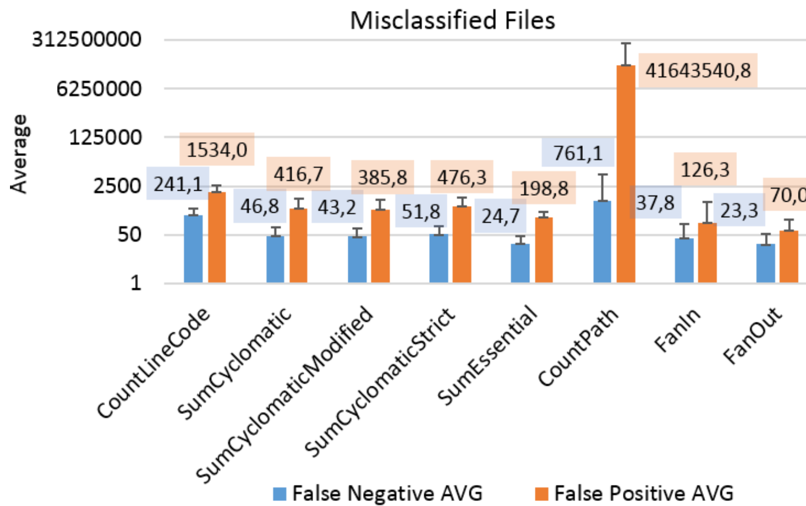


FIGURE 13. Software metrics' value for misclassified files.

TABLE 6. The most important software metrics with both Linux Kernel and Mozilla Firefox projects.

	Function-level Metrics		File-level Metrics	
	Xboost	Randomforest	Xboost	Randomforest
1	CountOutput	CountOutput	SumEssential	SumEssential
2	CountInput	CountInput	CountLineInactive	CountLineInactive
3	CountLineCodeDecl	CountLineCodeDecl	RatioCommentToCode	RatioCommentToCode
4	AltCountLineBlank	AltCountLineBlank	CountLineComment	CountLineComment
5	AltCountLineCode	MaxNesting	AltCountLineComment	AltCountLineComment
6	CountLineCode	CountLineBlank	AltCountLineBlank	AltCountLineBlank
7	CountLine		AltCountLineCode	AltAvgLineCode
8	CountPath		CountStmtDecl	AvgLineCode
9	CountLineCodeExe		CountLineBlank	MaxEssential
10			CountLinePreprocessor	
11			CountLineCodeDecl	

a dataset with higher diversity (including different types of software project), which is quite helpful for vulnerebility prediction of unseen code but still have a reasonable level of performance.

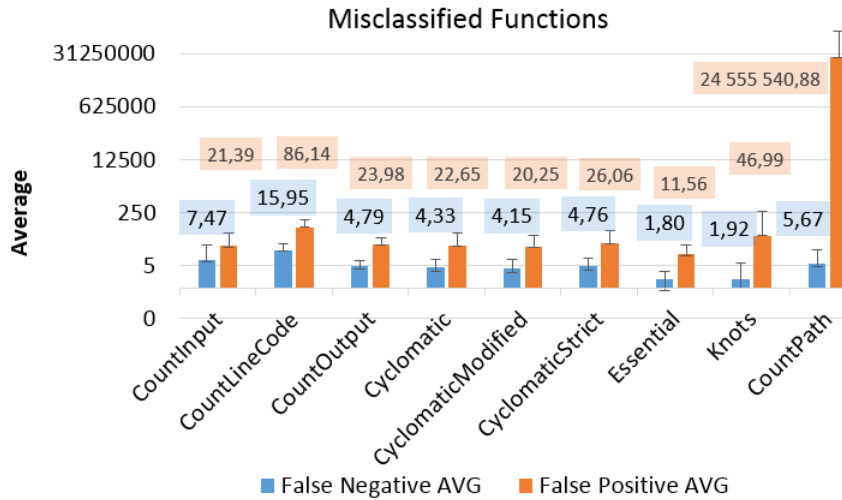


FIGURE 14. Software metrics' value for misclassified functions.

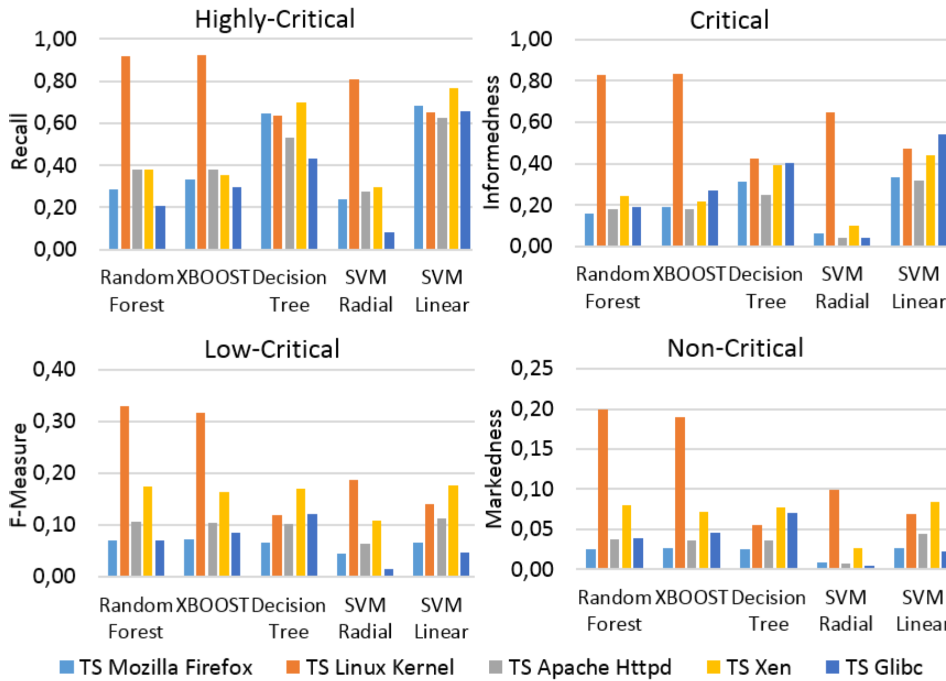


FIGURE 15. File-level inter-project cross-validation results (Linux Kernel data is used as training set).

VI. DISCUSSION AND THREATS TO VALIDITY

The main objective of this work was to perform a comprehensive experiment to demonstrate how effective software metrics combined with machine learning techniques can be to distinguish vulnerable from non-vulnerable code units in different application scenarios. Thus, we used several machine learning algorithms, several software project at both file and function levels, several application scenarios with different security concerns and several subset of software metrics to explore this idea. The main insights from the results are as follows:

- Machine learning algorithms using software metrics data can detect vulnerable code with a relatively high level of confidence for security-critical software systems (e.g., Recall and Informedness more than 0.8). However, a high number of false alarms makes the software metrics almost useless for low-critical or non-critical systems (i.e., response to **RQ1** defined in Section I).
- The larger and more complex a unit of code is, the more likely it is to have security issues. Thus, models built over software metrics that provide information regarding the structure and complexity of code can help to

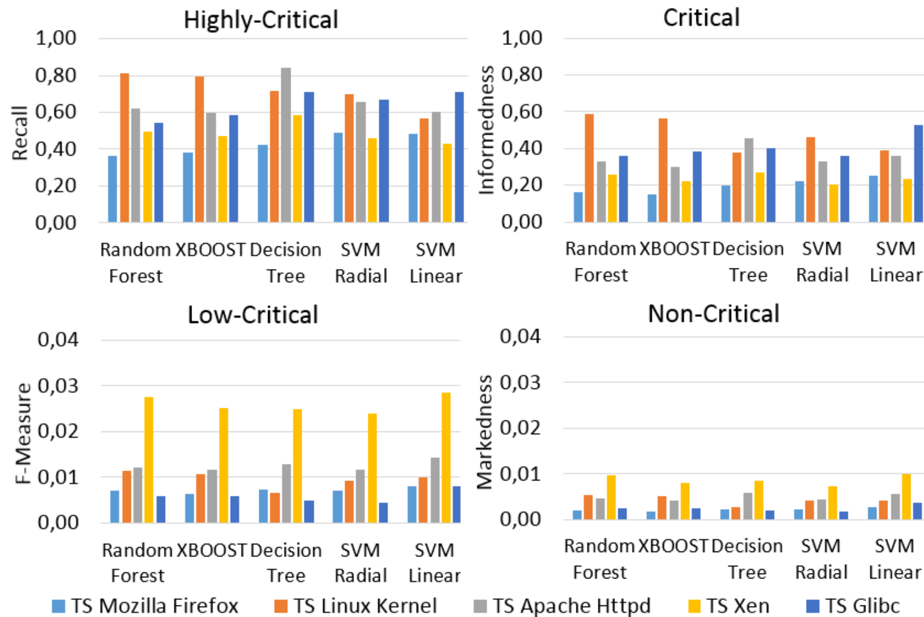


FIGURE 16. Function-level inter-project cross-validation results (Linux Kernel data is used as training set).

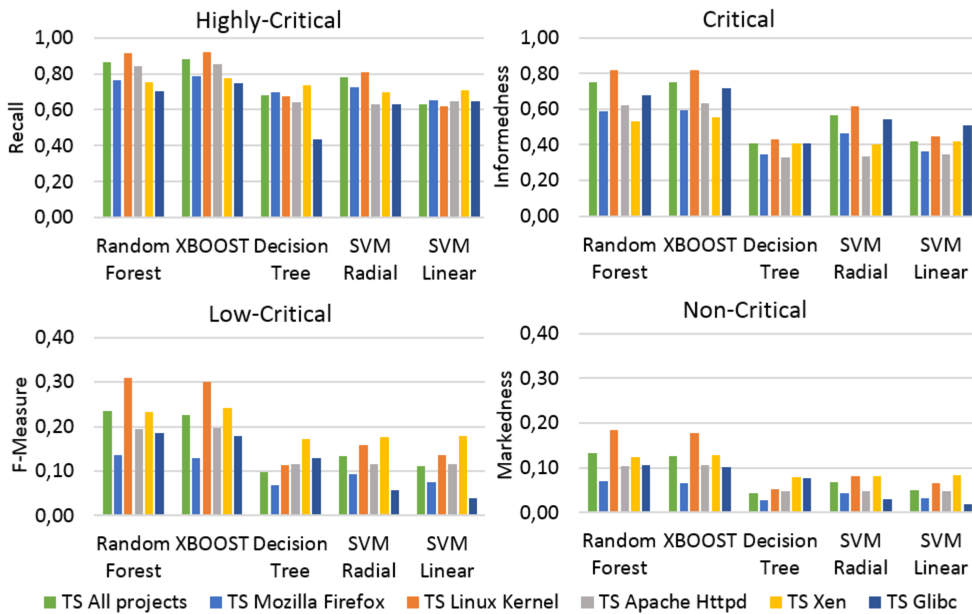


FIGURE 17. File-level generalization results.

predict vulnerabilities. Nevertheless, the high number of false alarms implies that software metrics are not sufficient to distinguish vulnerable and non-vulnerable code with high level of confidence and low cost (as false positives require resources to be verified). Moreover, software metrics are not able to indicate the exact place of the existing vulnerabilities. These limitations of software metrics imply that more evidences of low quality code (e.g., code smells or absence of security best practices) and deeper static (and dynamic) code analyses are required for building a high performance vulnerability detection/prediction tool.

- Undersampling the larger class of a dataset, in which a low percentage of data belongs to vulnerable code, helps to detect more vulnerable code, but with a higher false positive rate. Thus, the class distribution in the dataset to be used to train the classifiers is influenced by the application scenario.

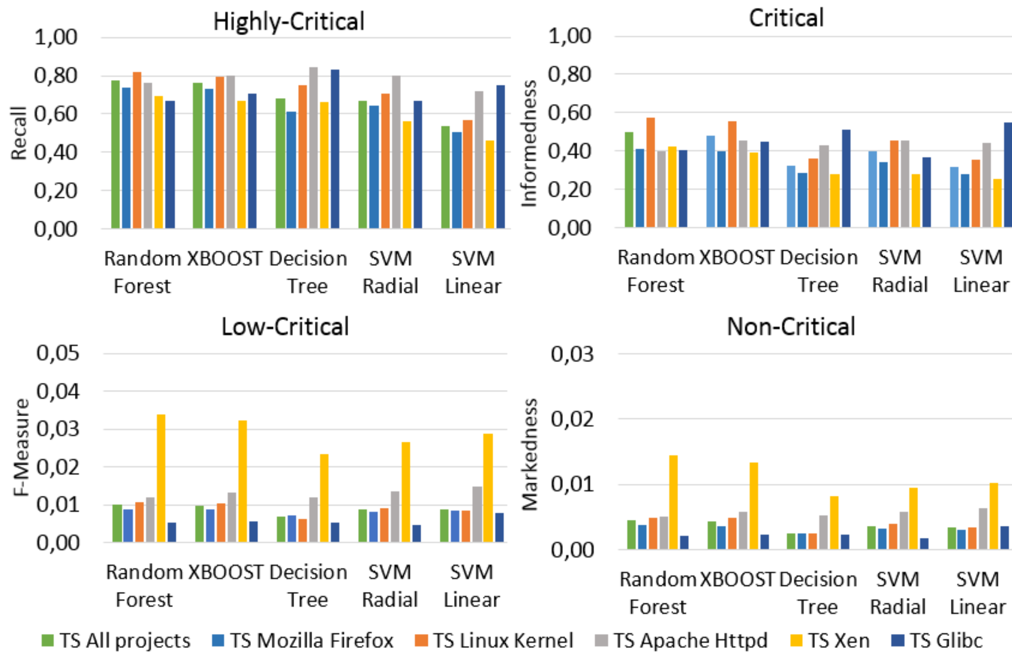


FIGURE 18. Function-level generalization results.

- Random Forest and Xboost were able to build more precise models than other algorithms to detect vulnerable code in a software project when data from the same project is used for training. In contrast, Decision Tree and Linear SVM seem to build more generalizable models, thus, give a better estimation when the data from another project is used to evaluate the model (i.e., response to **RQ3**).
- Considering particular application scenarios when building or choosing vulnerability detection tools is an important factor. We balanced the training set to build the classification models, which was helpful for detecting more vulnerable code at the cost of more false positive alarms, (suitable for highly critical systems where one prefers to detect more vulnerabilities, no matter how many false alarms are reported). In spite of that, balancing the training set for building the model was harmful for low or non-critical applications, making the models almost useless for these scenarios.
- In general, we cannot conclude that dimension reduction is or is not helpful to achieve better results in terms of performance. Indeed, the combination of metrics that leads to the best performance strongly depends on the machine learning algorithm. For example, Xboost achieves the best result when all metrics are used, while Decision Tree shows a better performance when the redundant file-level metrics are eliminated (i.e., response to **RQ2**).
- Misclassified vulnerable files seem to have different characteristics from misclassified non-vulnerable files in terms of structure and complexity. The same is true in the case of misclassified functions. Performing a deeper analysis of the source code in these files and functions can help to find new evidences or features that enable improving the performance of classifiers. To give an example, looking at Fig. 13 and Fig. 14, the average value of *CountPath* for missclassified vulnerable functions is 5.67, way lower than the average value for missclassified non-vulnereble functions. It means that, most of the missclassfied vulnerable functions are quite small and simple. By looking at their source code, we may find that only a single line of code with a sensitive operation (e.g., memcpy) made the code exploitable against attackers that could be simply avoided by adding a check before the operation. We intend to explore this deep analysis of the missclassified code in future.
- Our analysis of missclassified code shows that, although there is a group of code units that are missclassified by all the models, a bigger group of code units are missclassifed by only one or two models. Thus, it might be helpful to build a hybrid prediction model using several machine learning algorithms to lower the number of false alarms.
- The complex correlation between software metrics and between metrics and the existence of a vulnerability in the code makes it very difficult, if not impossible, to find a meaningful universal ranking of software metrics based on their importance in the prediction of vulnerable code (i.e., response to **RQ2**).
- The generalization assessment showed that the performance of classifiers is not degraded significantly when a dataset with diverse projects is used for training the

models. This can generally imply that the idea of using software metrics for the indication of vulnerable code can be generalized (i.e., response to **RQ4**).

We are aware that this experimental work has limitations that need to be taken into account when considering the insights above. Most of these threats to validity are related to the dataset used (selected due to the reasons mentioned in Section III). First, all the selected projects in the dataset are implemented in C/C++, and each programming language has its own characteristics in terms of security [92]. Consequently, some of the outcomes obtained from our analysis may not be representative for software implemented in other languages (e.g., Java).

The source of information regarding the vulnerabilities in the projects is limited to security reports. Consequently, the functions and files without reported vulnerabilities are not necessarily flawless. To build the classifier model, we followed a supervised approach, which considers that our dataset is completely labeled. However, although the records with vulnerabilities are (reliable) labeled, but the rest can be seen as being unlabeled. This way, semi-supervised approaches should be studied as alternative choices for such cases, where it is not trivial to verify the label of all records due to the size of the dataset and complexity of the code.

Although we used the well-known, commonly used, recommended, and representative machine learning algorithms, the number and diversity are still limited for a comprehensive analysis. Furthermore, the analysis for demonstrating the representativness of random samples as well as the analysis performed for the understanding the impact of class distribution are done over the source code of a single project. This may have some implications on the results obtained with the other projects.

We believe that main limitation and thread to the validity of this work and to the other similar works in the literature, comes from the fact that it is extremely difficult to build a dataset that is relatively balanced (i.e., having enough number of vulnerable code to prevent over-fitting), precisely labeled (i.e., all existing vulnerabilities identified for code labeled as vulnerable, and for code labeled as non-vulnerable assure that it is free of any vulnerability), and highly representative (i.e., covering a vast range of software projects implemented in different languages). Without such dataset, we will not be able to fully understand how effective software metrics can be to detect/predict vulnerable code for different application scenarios. Even if we imagine that such dataset already exists or can be built, it will be still a big challenge to build models that can guarantee a good performance with a low number of false alarms for previously unseen patterns of code.

VII. CONCLUSION AND FUTURE WORK

This paper presented a comprehensive study on the use of software metrics and machine learning algorithms for the detection/prediction of vulnerable code. The most important observation is that using machine learning algorithms on top of software metrics helps identifying vulnerable code units with relatively high level of confidence for security-critical software systems (where the focus is on detecting the maximum number of vulnerabilities, even if false positives are reported), but they are not helpful for low-critical or non-critical systems due to the relatively high number of false positive alarms reported when compared to the number of true positive cases (that bring an additional development cost frequently not affordable), which is mainly caused by the imbalanced nature of our dataset (and similar datasets used in other works).

According to our observations, insights and threats to the validity of the work, we can conclude that **software metrics are not sufficient evidence of security issues to be used solely for building detection/prediction models that are able to distinguish vulnerable code from non-vulnerable code with good performance and low vulnerability removal cost**. Moreover, due to the natural limitations of existing datasets for training and testing these models, it becomes even more difficult to precisely understand how effective software metrics can be to detect vulnerable code in different application scenarios. Based on this strong conclusion, we have two directions in front of us for future works.

The first direction will be focused on using other evidences rather than software metrics, like code smells [93], lack of security best practices in the code, alerts given by static code analysers, among others, to improve the detection/prediction models to produce less false alarms and try to find the location and type of vulnerabilities to provide some suggestions to developers for removing the detected or predicted vulnerabilities and improving the code. This requires a deep understanding of all (known) types of security issues and vulnerabilities as well as possible solutions for fixing them. In this scenario, we will still face the aforementioned limitations of models in new unknown situation. To address this issue, we can let the models to continuously adjust themselves by receiving feedback from developers of the code under development. After analysing the situation, and in the case of false alarms, the developers will send *feedback* to the analyser platform to *readjust the prediction model*. Otherwise, the suggestion is applied in the code either by writing new code or by changing (or removing) existing code. In practice, the prediction model will be continuously improved by new data generated from the code under development and feedback provided by developers. Here, the main challenge is to ensure that *i)* the software functionality remains after applying the changes; and *ii)* the changes do not introduce a new bug or vulnerability in the system.

The second direction will be focused on using software metrics not for predicting or detecting vulnerabilities but for assessing the trustworthiness of the code and warn the developers about their untrustworthy (insecure) code units. In a previous work [94], we proposed a trustworthiness model directly by using a group of software metrics that were weighted based on the scores given by a classification model. Despite the merit of that work, the results of the current work show that such model cannot be generalized, since it is

almost impossible to find a meaningful universal ranking of software metrics, based on their importance in the prediction of vulnerable code, to be used for any kind of software. For this reason, we suggest to build a trustworthiness model, not directly based on software metrics, but based on the classification results of several machine learning algorithms that are trained using software metrics. This solution does not find vulnerable code, but may be able to warn developers about the units of code that seem to be more untrustworthy. By assigning a trustworthiness score to each unit of code, it is up to the developers to decide what part of the code needs more attention (depending on the criticality of the application and the available resources), thus being suitable for any application scenarios.

REFERENCES

[1] G. McGraw, *Software Security: Building Security*, vol. 1. Reading, MA, USA: Addison-Wesley, 2006.

[2] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2000, pp. 1–15.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[4] I. A. Elia, N. Antunes, N. Laranjeiro, and M. Vieira, "An analysis of OpenStack vulnerabilities," in *Proc. 13th Eur. Dependable Comput. Conf. (EDCC)*, Sep. 2017, pp. 129–134.

[5] D. Galin, *Software Quality Assurance: From Theory to Implementation*. London, U.K.: Pearson, 2004.

[6] A. Cavoukian, "Privacy by design—The 7 foundational principles–implementation and mapping of fair information practices," Inf. Privacy Commissioner Ontario, Toronto, ON, Canada, Tech. Rep., 2009. [Online]. Available: https://www.ipc.on.ca

[7] M. A. Cusumano, "Who is liable for bugs and security flaws in software?" *Commun. ACM*, vol. 47, no. 3, pp. 25–27, Mar. 2004.

[8] B. Chess and G. McGraw, "Static analysis for security," *IEEE Secur. Privacy Mag.*, vol. 2, no. 6, pp. 76–79, Nov. 2004.

[9] K. Senthamil and A. Murugan, "Analysis of vulnerability detection tool for Web services," *Int. J. Eng. Technol.*, vol. 7, pp. 773–778, May 2018.

[10] J. A. Wang, H. Wang, M. Guo, and M. Xia, "Security metrics for software systems," in *Proc. 47th Annu. Southeast Regional Conf.*, 2009, p. 47.

[11] Y. Shin and L. Williams, "Is complexity really the enemy of software security?" in *Proc. 4th ACM Workshop Qual. Protection (QoP)*, 2008, pp. 47–50.

[12] H. Alves, B. Fonseca, and N. Antunes, "Software metrics and security vulnerabilities: Dataset and exploratory study," in *Proc. 12th Eur. Dependable Comput. Conf. (EDCC)*, Sep. 2016, pp. 37–44.

[13] R. Kumar, S. A. Khan, and R. A. Khan, "Durable security in software development: Needs and importance," *CSI Commun.*, vol. 10, pp. 34–36, Oct. 2015.

[14] Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl, "Developers need support, too: A survey of security advice for software developers," in *Proc. IEEE Cybersecurity Develop. (SecDev)*, Sep. 2017, pp. 22–26.

[15] G. Disterer, *27001 and 27002 for Information Security Management*, Iso/IEC document 27000, 2013.

[16] O. Potii, O. Illiashenko, and D. Komin, "Advanced security assurance case based on ISO/IEC 15408," in *Proc. Int. Conf. Dependability Complex Syst.* Cham, Switzerland: Springer, 2015, pp. 391–401.

[17] M. Chemuturi, *Mastering Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers*. J. Ross Publishing, 2010.

[18] D. Heimann, *IEEE Standard 730-2014 Software Quality Assurance Processes*, IEEE Comput. Soc., New York, NY, USA, IEEE Standard 730:2014, 2014.

[19] K. Turpin, "Owasp secure coding practices-quick reference guide," Tech. Rep., 2010. [Online]. Available: https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/

[20] J. Marcil, "OWASP ISO IEC 27034 application security controls project," in *OWASP-Open Web Application Security Project*. Bel Air, MD, USA: OWASP Foundation, 2014. [Online]. Available: https://owasp.org

[21] L. Poulin and B. Guay, *Application Security-Overview*, 20.1 Kyoto, ISO/IEC document 27034, 2008, p. 29.

[22] H.-W. Jung, S.-G. Kim, and C.-S. Chung, "Measuring software product quality: A survey of ISO/IEC 9126," *IEEE Softw.*, vol. 21, no. 05, pp. 88–92, Sep. 2004.

[23] *Information Technology-Security Techniques-Evaluation Criteria for it Security*, ISO Standard 15408-1: 2009, 2009.

[24] G. Disterer, "ISO/IEC 27000, 27001 and 27002 for information security management," *J. Inf. Secur.*, vol. 04, no. 02, pp. 92–100, 2013.

[25] K. Beckers, I. Côté, S. Fenz, D. Hatebur, and M. Heisel, "A structured comparison of security standards," in *Engineering Secure Future Internet Services and Systems*. Springer, 2014, pp. 1–34.

[26] L. Shan, B. Sangchoolie, P. Folkesson, J. Vinter, E. Schoitsch, and C. Loiseaux, "A survey on the applicability of safety, security and privacy standards in developing dependable systems," in *Proc. Int. Conf. Comput. Saf., Rel., Secur.* Cham, Switzerland: Springer, Cham, 2019, pp. 74–86.

[27] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Softw.*, vol. 19, no. 1, pp. 42–51, Jan./Feb. 2002.

[28] M. Graff and K. R. Van Wyk, *Secure Coding: Principles and Practices*. Newton, MA, USA: O'Reilly Media, Inc., 2003.

[29] G. Campbell and P. P. Papapetrou, *SonarQube Action*. Shelter Island, NY, USA: Manning Publications Co., 2013.

[30] P. De Cremer, N. Desmet, M. Madou, and B. De Sutter, "Sensei: Enforcing secure coding guidelines in the integrated development environment," *Software: Pract. Exper.*, vol. 50, no. 9, pp. 1682–1718, Sep. 2020.

[31] G. Ouffoué, F. Zaïdi, and A. R. Cavalli, "Attack tolerance for services-based applications in the cloud," in *Proc. IFIP Int. Conf. Test. Softw. Syst.* Cham, Switzerland: Springer, 2019, pp. 242–258.

[32] M. Abdlhamed, K. Kifayat, Q. Shi, and W. Hurst, "Intrusion prediction systems," in *Information Fusion for Cyber-Security Analytics*. Cham, Switzerland: Springer, 2017, pp. 155–174.

[33] G. Ouffoue, A. M. Ortiz, A. R. Cavalli, W. Mallouli, J. Domingo-Ferrer, D. Sanchez, and F. Zaidi, "Intrusion detection and attack tolerance for cloud environments: The CLARUS approach," in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, Jun. 2016, pp. 61–66.

[34] Y. Deswarte, L. Blain, and J.-C. Fabre, "Intrusion tolerance in distributed computing systems," in *Proc. IEEE Comput. Soc. Symp. Res. Secur. Privacy*, May 1991, pp. 110–121.

[35] R. Di Pietro and L. V. Mancini, Eds., *Intrusion Detection Systems*, vol. 38. Springer, 2008.

[36] H. Assal and S. Chiasson, "'Think secure from the beginning' A survey with software developers," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2019, pp. 1–13.

[37] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Secur. Privacy*, vol. 3, no. 1, pp. 84–87, Jan. 2005.

[38] A. Neto and M. Vieira, "Selecting secure Web applications using trustworthiness benchmarking," *Int. J. Dependable Trustworthy Inf. Syst.*, vol. 2, no. 2, pp. 1–6, 2011.

[39] Z. Li and Y. Shao, "A survey of feature selection for vulnerability prediction using feature-based machine learning," in *Proc. 11th Int. Conf. Mach. Learn. Comput. (ICMLC)*, 2019, pp. 36–42.

[40] J. E. Gaffney, "Metrics in software quality assurance," in *Proc. ACM Conf. (ACM)*, 1981, pp. 126–130.

[41] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, Aug. 1994.

[42] L. Rosenberg, T. Hammer, and J. Shaw, "Software metrics and reliability," in *9th Int. Symp. Softw. Rel. Eng.*, 1998, pp. 1–8.

[43] L. C. Briand, J. Wüst, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, May 2000.

[44] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[45] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," 2018, *arXiv:1807.06756*. [Online]. Available: http://arxiv.org/abs/1807.06756

[46] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, ''Automated vulnerability detection in source code using deep representation learning,'' in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 757–762.

[47] Meiliana, S. Karim, H. L. H. S. Warnars, F. L. Gaol, E. Abdurachman, and B. Soewito, ''Software metrics for fault prediction using machine learning approaches: A literature review with PROMISE repository dataset,'' in *Proc. IEEE Int. Conf. Cybern. Comput. Intell. (CyberneticsCom)*, Nov. 2017, pp. 19–23.

[48] S. Moshtari, A. Sami, and M. Azimi, ''Using complexity metrics to improve software security,'' *Comput. Fraud Secur.*, vol. 2013, no. 5, pp. 8–17, May 2013.

[49] M. Alenezi and M. Zarour, ''Software vulnerabilities detection based on security metrics at the design and code levels: empirical findings,'' *J. Eng. Technol.*, vol. 6, no. 1, pp. 570–583, 2018.

[50] S. M. Ghaffarian and H. R. Shahriari, ''Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey,'' *ACM Comput. Surv.*, vol. 50, no. 4, p. 56, Nov. 2017.

[51] I. Chowdhury and M. Zulkernine, ''Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities,'' *J. Syst. Archit.*, vol. 57, no. 3, pp. 294–313, Mar. 2011.

[52] Y. Shin and L. Williams, ''An initial study on the use of execution complexity metrics as indicators of software vulnerabilities,'' in *Proc. 7th Int. Workshop Softw. Eng. Secure Syst. (SESS)*, 2011, pp. 1–7.

[53] X. Shen, ''Predicting vulnerable files by using machine learning method,'' M.S. thesis, Dept. Elect. Eng., Math. Comput. Sci. (EWI), Delft Univ. Technol., Delft, The Netherlands, 2018.

[54] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, ''Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities,'' *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov. 2011.

[55] J. Ren, Z. Zheng, Q. Liu, Z. Wei, and H. Yan, ''A buffer overflow prediction approach based on software metrics and machine learning,'' *Secur. Commun. Netw.*, vol. 2019, pp. 1–13, Mar. 2019.

[56] K. Z. Sultana, B. J. Williams, and A. Bosu, ''A comparison of nano-patterns vs. Software metrics in vulnerability prediction,'' in *Proc. 25th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2018, pp. 355–364.

[57] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, ''Software metrics as indicators of security vulnerabilities,'' in *Proc. IEEE 28th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2017, pp. 216–227.

[58] H. Alves, B. Fonseca, and N. Antunes, ''Experimenting machine learning techniques to predict vulnerabilities,'' in *Proc. 7th Latin-American Symp. Dependable Comput. (LADC)*, Oct. 2016, pp. 151–156.

[59] Y. Shin, ''Exploring complexity metrics as indicators of software vulnerability,'' in *Proc. Int. Doctoral Symp. Empirical Soft. Eng.*, 2008, p. 3.

[60] Y. Shin and L. Williams, ''Can traditional fault prediction models be used for vulnerability prediction?'' *Empirical Softw. Eng.*, vol. 18, no. 1, pp. 25–59, Feb. 2013.

[61] N. A. Henrique Alves and B. Fonseca. (2016). *A Dataset of Source Code Metrics and Vulnerabilities*. [Online]. Available: https://eden.dei.uc.pt/~nmsa/metrics-dataset/

[62] SciTools. (2017). *Understand Static Code Analysis Tool*. [Online]. Available: https://support.scitools.com/t/what-metrics-does-undertand-have/66

[63] H. Liu, E. R. Dougherty, J. G. Dy, K. Torkkola, E. Tuv, H. Peng, C. Ding, F. Long, M. Berens, and L. Parsons, ''Evolving feature selection,'' *IEEE Intell. Syst.*, vol. 20, no. 6, pp. 64–76, Nov. 2005.

[64] M.-R. Feizi-Derakhshi and M. Ghaemi, ''Classifying different feature selection algorithms based on the search strategies,'' in *Proc. Int. Conf. Mach. Learn., Electr. Mech. Eng.*, 2014, pp. 17–21.

[65] J. Benesty, J. Chen, Y. Huang, and I. Cohen, ''Pearson correlation coefficient,'' in *Noise Reduction in Speech Processing*. Berlin, Germany: Springer, 2009, pp. 1–4.

[66] L. Myers and M. J. Sirois, ''S pearman Correlation Coefficients, Differences between,'' *Wiley StatsRef: Statist. Reference Online*, to be published, doi: 10.1002/0471667196.ess5050.

[67] A. Bommert, X. Sun, B. Bischl, J. Rahnenführer, and M. Lang, ''Benchmark for filter methods for feature selection in high-dimensional classification data,'' *Comput. Statist. Data Anal.*, vol. 143, Mar. 2020, Art. no. 106839.

[68] L. Yu and H. Liu, ''Efficient feature selection via analysis of relevance and redundancy,'' *J. Mach. Learn. Res.*, vol. 5, no. 10, pp. 1205–1224, 2004.

[69] A. Wang, N. An, J. Yang, G. Chen, L. Li, and G. Alterovitz, ''Wrapper-based gene selection with Markov blanket,'' *Comput. Biol. Med.*, vol. 81, pp. 11–23, Feb. 2017.

[70] N. V. Chawla, N. Japkowicz, and A. Kotcz, ''Special issue on learning from imbalanced data sets,'' *ACM SIGKDD Explor. Newslett.*, vol. 6, no. 1, pp. 1–6, 2004.

[71] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard, ''A study of the behavior of several methods for balancing machine learning training data,'' *ACM SIGKDD Explor. Newslett.*, vol. 6, no. 1, pp. 20–29, Jun. 2004.

[72] A. Estabrooks, T. Jo, and N. Japkowicz, ''A multiple resampling method for learning from imbalanced data sets,'' *Comput. Intell.*, vol. 20, no. 1, pp. 18–36, Feb. 2004.

[73] D. Landgrebe, ''A survey of decision tree classifier methodology,'' *IEEE Trans. Syst., Man Cybern.*, vol. 21, no. 3, pp. 660–674, May 1991.

[74] L. Breiman, ''Random forests,'' *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[75] M. Awad and R. Khanna, ''Support vector machines for classification - efficient learning machines: Theories, concepts, and applications for engineers and system designers,'' in *Efficient Learning Machines*. New York, NY, USA: Apress, 2015, pp. 39–66.

[76] B. E. Boser, I. M. Guyon, and V. N. Vapnik, ''A training algorithm for optimal margin classifiers,'' in *Proc. 5th Annu. Workshop Comput. Learn. Theory (COLT)*, 1992, pp. 144–152.

[77] S. Dreiseitl and L. Ohno-Machado, ''Logistic regression and artificial neural network classification models: A methodology review,'' *J. Biomed. Informat.*, vol. 35, nos. 5–6, pp. 352–359, Oct. 2002.

[78] D. Westreich, J. Lessler, and M. J. Funk, ''Propensity score estimation: Neural networks, support vector machines, decision trees (CART), and meta-classifiers as alternatives to logistic regression,'' *J. Clin. Epidemiology*, vol. 63, no. 8, pp. 826–833, Aug. 2010.

[79] T. Chen and C. Guestrin, ''XGBoost: A scalable tree boosting system,'' in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 785–794.

[80] S. Georganos, T. Grippa, S. Vanhuysse, M. Lennert, M. Shimoni, and E. Wolff, ''Very high resolution object-based land use–land cover urban classification using extreme gradient boosting,'' *IEEE Geosci. Remote Sens. Lett.*, vol. 15, no. 4, pp. 607–611, Apr. 2018.

[81] R. E. Schapire, ''The boosting approach to machine learning: An overview,'' in *Nonlinear Estimation and Classification*. New York, NY, USA: Springer, 2003, pp. 149–171.

[82] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira, ''Benchmarking static analysis tools for Web security,'' *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 1159–1175, Sep. 2018.

[83] N. Antunes and M. Vieira, ''On the metrics for benchmarking vulnerability detection tools,'' in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2015, pp. 505–516.

[84] C. H. Yu, ''Resampling methods: concepts, applications, and justification,'' *Practical Assessment, Res., Eval.*, vol. 8, no. 1, p. 19, 2002.

[85] R. C. Team. (2017). *The R Project in Statistical Computing*. [Online]. Available: https://www.r-project.org/foundation/

[86] M. Kuhn *et al.*, ''Package 'caret,''' *R J.*, 2020. [Online]. Available: http://topepo.github.io/caret/index.html

[87] A. Liaw and M. Wiener. (2018). *The R Random Forest Package*. [Online]. Available: https://cran.r-project.org/web/packages/randomForest/index.html

[88] D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel, F. Leisch, C.-C. Chang, and C.-C. Lin. (2019). *e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071)*. [Online]. Available: https://cran.r-project.org/web/packages/e1071/index.html

[89] H. Wickham, R. Francois, L. Henry, K. Muller, and RStudio. (2019). *dplyr: A Grammar of Data Manipulation*. [Online]. Available: https://cran.r-project.org/web/packages/dplyr/index.html

[90] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, ''Orthogonal defect classification–A concept for in-process measurements,'' *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, Nov. 1992.

[91] H. Kaur and P. Jai, ''Comparing detection ratio of three static analysis tools,'' *Int. J. Comput. Appl.*, vol. 124, no. 13, pp. 35–40, Aug. 2015.

[92] S. Turner, ''Security vulnerabilities of the top ten programming languages: C, Java, C++, Objective-C, C#, PHP, Visual Basic, Python, Perl, and Ruby,'' *J. Technol. Res.*, vol. 5, p. 1, Oct. 2014.

[93] A. Cairo, G. Carneiro, and M. Monteiro, "The impact of code smells on software bugs: A systematic literature review," *Information*, vol. 9, no. 11, p. 273, Nov. 2018.

[94] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "An approach for trustworthiness benchmarking using software metrics," in *Proc. IEEE 23rd Pacific Rim Int. Symp. Dependable Comput. (PRDC)*, Dec. 2018, pp. 84–93.

NÁDIA MEDEIROS is currently pursuing the Ph.D. degree with the Centre for Informatics and Systems (CISUC), Department of Informatics Engineering, University of Coimbra. She is also a Researcher with the Centre for Informatics and Systems (CISUC), Software and Systems Engineering Group (SSE), 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 these fields.

NAGHMEH IVAKI (Member, IEEE) received the Ph.D. degree from the University of Coimbra, Portugal. She is currently a Postdoctoral Researcher and a Full Member with the Centre for Informatics and Systems (CISUC), Software and Systems Engineering Group (SSE), 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 25 peer-reviewed publications and participated in several national and international research projects.

PEDRO COSTA is currently a Professor of computer science with the Polytechnic Institute of Coimbra and a Researcher with the Center for Informatics and Systems, University of Coimbra, where he integrates the Software and Systems Engineering Group (SSE). He is also the President of the Coimbra Business School (ISCAC), Portugal. His research interests include dependable systems, dependability and security assessment and benchmarking, software fault tolerance, and software implemented fault injection (SWIFI).

MARCO VIEIRA (Member, IEEE) is currently a Full Professor with the University of Coimbra, Portugal. His research interests include dependability and security assessment and benchmarking, fault injection, software processes, and software quality assurance, subjects in which he has authored or coauthored more than 200 papers in refereed conferences and journals. He has participated and coordinated several research projects, both at the national and European levels. He has served on program committee of the major conferences for the dependability area and acted as a referee for many international conferences and journals in the dependability and security areas.