Just-in-Time Software Vulnerability Detection: Are We There Yet?

Francesco Lomio[a], Emanuele Iannone[b], Andrea De Lucia[b], Fabio Palomba[b], Valentina Lenarduzzi[c]

[a] *Tampere University, Finland*
[b] *SeSa Lab — Department of Computer Science, University of Salerno, Italy*
[c] *University of Oulu, Finland*

Abstract

Background. Software vulnerabilities are weaknesses in source code that might be exploited to cause harm or loss. Previous work has proposed a number of automated machine learning approaches to detect them. Most of these techniques work at release-level, meaning that they aim at predicting the files that will potentially be vulnerable in a future release. Yet, researchers have shown that a commit-level identification of source code issues might better fit the developer's needs, speeding up their resolution. *Objective.* To investigate how currently available machine learning-based vulnerability detection mechanisms can support developers in the detection of vulnerabilities at commit-level. *Method.* We perform an empirical study where we consider nine projects accounting for 8,991 commits and experiment with eight machine learners built using process, product, and textual metrics. *Results.* We point out three main findings: (1) basic machine learners rarely perform well; (2) the use of ensemble machine learning algorithms based on boosting can substantially improve the performance; and (3) the combination of more metrics does not necessarily improve the classification capabilities. *Conclusion.* Further research should focus on just-in-time vulnerability detection, especially with respect to the introduction of smart approaches for feature selection and training strategies.

Keywords: Software Vulnerabilities; Machine Learning; Empirical SE.

1. Introduction

Software security plays a crucial role in modern software development [1]. In software engineering terms, this has to do with the implementation of programs that can continue working under malicious circumstances [2]. Specifically, the source code should be designed to be resilient to external attacks: unfortunately, software vulnerabilities represent threats to security that may potentially be exploited by externals to cause loss of data, privilege escalation, race conditions, and other undesired effects that may affect the source code [3, 4].

The research community has been addressing the problem of vulnerabilities under different perspectives, by proposing empirical studies aiming at characterizing them and their impact on source code [5, 6, 7], but more importantly by devising automated techniques that could support their identification [8, 9, 10].

Most of the approaches defined so far are based on source code and/or dynamic analysis [11, 12, 13, 14], symbolic execution [15, 16, 17], and fuzz-testing [18, 19]. Some of them are also implemented within automated tools, e.g., SonarQube[1] and Eclipse Steady[2] that are widely adopted in practice [20].

Despite the research and industrial effort spent so far for building techniques and tools able to identify software vulnerabilities, the current solutions are still rarely effective in practice as they suffer from high false positive rates and/or scalability issues [21, 22, 23].

For these reasons, the research around vulnerability detection is still highly active. The last years have seen a growing interest in the application of artificial intelligence algorithms to software security [24, 25]. Techniques based on machine learning, in particular, have reached promising results: starting from a set of vulnerability data collected through the change history analysis of files over previous releases of an application, these techniques train machine learning algorithms (e.g., Decision Tree) in order to predict the likelihood of new, unseen source code files to be affected by vulnerabilities in future releases [26].

While the performance reported in previous studies [27, 28, 29, 30, 31] highlighted the suitability of machine learning approaches to predict vulnerabilities on future releases, it is still unclear how these approaches support developers in finding the exact location of the vulnerable

Email addresses: francesco.lomio@tuni.fi (Francesco Lomio), eiannone@unisa.it (Emanuele Iannone), adelucia@unisa.it (Andrea De Lucia), fpalomba@unisa.it (Fabio Palomba), valentina.lenarduzzi@oulu.fi (Valentina Lenarduzzi)

[1] SonarQube: https://www.sonarqube.org.

[2] Eclipse Steady: https://projects.eclipse.org/proposals/eclipse-steady.

code. As a matter of fact, traditional vulnerability predictions [29, 30, 27, 32] would produce a large set of potentially vulnerable files or binaries, that should be manually inspected to establish the actual presence of the flaw, requiring a non-negligible amount of extra work. Moreover, such a task requires the selection of the most appropriate group of developers that can comprehend the rationale behind the last changes applied to the files. These limitations calls for novel solutions that better suits real case scenarios. Specifically, contemporary pull-based development practices [33] make long-term recommendations, like those given by release-based predictions, not really suitable [34]. Shorter-term recommendations, also known as *just-in-time* or *commit-level* predictions, should be preferred instead as they allow developers to receive an immediate feedback on the newly committed work and improve code quality while having the context of the modification still fresh in mind [35, 36]. In addition, techniques able to work at this granularity become not only suitable at commit-time, but also while developers perform code review [37]. As a consequence of these recent advances, vulnerability detection mechanisms should be re-assessed at a lower granularity.

Hence, this paper proposes an empirical investigation into the performance of just-in-time software vulnerability detection techniques. We mine nine JAVA projects available in the *National Vulnerability Database* (NVD)[3] in order to collect known vulnerabilities that affected them during their change history. Afterwards, we experiment with eight machine learning algorithms that we train using three different sets of features based on code, change, and textual metrics—both algorithms and features were previously employed in the context of vulnerability detection research. In addition, we employ a set of machine learning engineering steps [38] aiming at improving the performance of the experimented models, such as dropping correlated features [39], balancing the dataset [40], and tuning hyper-parameters [41].

The results of our study reveal a number of findings. In the first place, we observe that basic machine learning algorithms, e.g., SUPPORT VECTOR MACHINE, have low performance when applied for the task of detecting vulnerabilities at commit-level, in contrast with previous work on vulnerability prediction. Moreover, the use of ensemble techniques do not necessarily provide benefits, even tough approaches based on boosting, like ADABOOST, seems promising and might be further investigated. Finally, we point out the limitations of existing metrics: for instance, we observe that previously devised textual metrics based on a raw *bag-of-words* source code representation lead the machine learners to have high variability and low prediction accuracy.

To sum up, we provide the following contributions:

1. Empirical evidence on the limited capabilities of commit-level vulnerability prediction models built using traditional techniques without proper setup;
2. A set of insights into the likely causes of failure of the current solutions, which forms the future research direction on the matter;
3. An online appendix[4] providing all data and scripts used to conduct our study and that can be used by the research community to replicate and build upon our empirical study.

Structure of the paper. Section 2 discusses the related literature and motivates our work. In Section 3 we report the methodology employed to address our goals, while Section 4 analyzes the achieved results. The key implications of the study are presented in Section 5. The discussion of the threats to validity and how we mitigated them is reported in Section 6. Finally, Section 7 concludes the paper and outlines our future research agenda on the matter.

2. Related Work

Research on software vulnerability prediction models (VPMs) mainly focused on identifying the best set of predictors correlated with the presence of vulnerabilities. Almost all works involved software product metrics directly computed on the source or binary files, such as size (e.g., Lines of Code) or structural metrics [53]. Among these, complexity metrics (e.g., McCabe's Cyclomatic Complexity [54]) are the ones that have received more attention. Shin et al. [55, 28, 48], in the context of MOZILLA FIREFOX, found a strong positive correlation between the number of decisions in the code and the vulnerability-proneness of a file. Specifically, the VPMs—built using complexity metrics as predictors—achieve higher precision scores if the predictions are restricted to the top vulnerable files only, hinting that the files that were subject to many vulnerabilities in the past have high complexity values. This finding is further confirmed in other studies [46, 29, 49, 32]. Similarly, coupling and cohesion metrics have been shown to be, respectively, positively and negatively correlated with vulnerabilities, corroborating the common wisdom that poor quality code raised the risk of introducing flaws [46]. Moreover, Nguyen and Tran [45] exploited a set of metrics extracted from the Component Dependency Graphs (CDG) to predict vulnerable C++ files in JS ENGINE of FIREFOX, observing an improvement in both accuracy and recall with respect to models built considering complexity metrics only. Neuhaus et al. [42] found a correlation between the number of imports and functions with vulnerabilities in C functions, hinting their usefulness in a VPM. In particular, they devised a Support Vector Machine (SVM) relying on the number of past vulnerabilities on the imported C files in the context

[3]The National Vulnerability Database: `https://nvd.nist.gov/`.

[4]Our online appendix: `https://figshare.com/s/0ef0f484a058e2297df4`.

Table 1: Comparison with previous works concerning vulnerability prediction models. The focus is on the granularity level (i.e., the components that is subject to the predictions), the set of metrics used as predictors, the involved systems and the mined vulnerability data sources.

Study	Granularity	Predictors/Features	Context	Data Sources
Neuhaus et al. [42]	Function	Past vulnerable imports	Firefox	MFSA
Sultana et al. [43]	Class/Method	Product metrics	4 Java systems	Vendor Advisories
Zimmermann et al. [29]	Binary	Process and Product metrics	Windows Vista	NVD
Theisen et al. [30]	Binary/File	Past crashes, Process and Product metrics	Windows 8	Microsoft Error Reporting
Theisen and Williams [31]	Binary/File	Past crashes, Process and Product metrics, Bag-of-words	Firefox	MFSA
Morrison et al. [44]	Binary/File	Product metrics	Windows 7 and 8	NVD
Nguyen and Tran [45]	File	Product metrics	Firefox JS Engine	MFSA, Bugzilla, NVD
Chowdhury et al. [46]	File	Product metrics	Firefox	MFSA, Bugzilla
Smith and Williams [47]	File	SQL Hotspots, LOC	WordPress, WakkaWiki	Bugzilla
Shin et al. [28]	File	Process and Product metrics	Firefox, RHEL	MFSA, RHSR, Bugzilla
Shin et al. [48]	File	Past faults, Process and Product metrics	Firefox	Bugzilla
Scandariato et al. [27]	File	Bag-of-words	20 Android apps	Fortify SCA
Walden et al. [49]	File	Product metrics, Bag-of-words	3 PHP systems	Vendor Security Advisories, NVD
Zhang et al. [50]	File	Product metrics, Bag-of-words	3 PHP systems	Dataset from [49].
Jimenez et al. [32]	File	Bag-of-words, Process and Product metrics	Linux, OpenSSL and Wireshark	NVD
Perl et al. [51]	Commit	Process and Product metrics	66 C/C++ systems	CVE DB
Yang et al. [52]	Commit	Process and Product metrics	Firefox	MFSA
Our study	**Commit**	**Process and Product metrics, Bag-of-words**	**9 Java systems**	**NVD**

of Mozilla Firefox achieving a high precision of 70%, at the cost of a lower recall of 45%. Furthermore, Scandariato et al. [27] were the first to investigate on the predictive power of text mining techniques. Namely, they used the *bag-of-words* method [56, 57] to extract the most frequent terms (i.e., words) from source code Java files to predict the presence of vulnerabilities on 20 Android apps. They managed to score a high performance in within-project predictions (i.e., making prediction on files belonging to the same project in which the model was trained), but failing in cross-project scenario (i.e., making prediction on files not belonging to the projects in which the model was trained), as further confirmed by Walden et al. [49]. Later, Zhang et al. [50] combined the above bag-of-words method with traditional product metrics, achieving higher F-measure value with respect to the VPM in [49]. On the other hand, Zimmermann et al. [29] analyzed the impact of organizational (e.g., the number of developers) and code churn (i.e., the rate of changes applied to binaries) metrics to vulnerabilities in Windows Vista, achieving high precision but low recall, in line with the findings of later studies [28, 32, 51, 58]. Smith and Williams [47] tested the usage of warnings of possible SQL Injections as predictors in two VPMs for WordPress and WakkaWiki, finding a positive correlation with many vulnerability types—other than SQL Injection. All the above findings are mixed together in the study of Theisen and Williams [31], in which the authors claimed that the best prediction models are the one encompassing many different set of metrics (namely, product, process, text metrics and past faults).

Regarding the model selection, vulnerability prediction have been based on different supervised machine learning models, such as Decision Trees [28, 27, 50, 32, 31], Support Vector Machines (SVM) [42, 45, 29, 27, 30, 51, 44], Naïve Bayes [45, 28, 27, 44, 50, 31] and Random Forests [28, 27, 49, 44, 50, 32, 31]. Among these, Naïve Bayes resulted in higher recall values (which means lower false negative rate), while Random Forests regularly score high precision (meaning low false positive rate) in different contexts. Such models are also popular in similar tasks, e.g., defect [48, 59] and exploitability prediction [60].

Most studies have been conducted on predicting vulnerabilities at source code file level [45, 46, 47, 55, 28, 27, 49, 50, 32], which means that the VPM tells whether a given file is or is not affected by a vulnerability. In such a scenario, the developers can invest their effort on inspecting and testing the problematic files with dedicated attention. The same concept is applied for VPMs working on binary files [29, 30, 31, 44], which contain machine code produced by a compiler. Neuhaus et al. [42] designed a tool, Vulture, that predicts the vulnerabilities in C/C++ functions, whereas Sultana et al. [43] do this on Java methods, instead. To the best of our knowledge, only few works have considered the predictions at commit-level. Perl et al. [51] devised a method for obtaining the vulnerability-contributing commit on 66 C/C++

open-source projects. They essentially relied on the `git blame` command that reaches the commits that changed last the deleted lines of a public fixing commit of known vulnerabilities reported in NVD. Then, they labelled the most blamed commit as a vulnerability-contributing commit. Finally, they trained a Support Vector Machine on this dataset, outperforming the detection capabilities of equivalent static analysis tools. The entire pipeline was replicated some years later by Riom et al. [61], in which the authors, among other things, delve into the possibility to improve VPM provided by Perl et al. [51] by experimenting on a different feature set containing metrics capturing more security-related aspects—e.g., the number of `sizeof` operators, which are known to be closely linked to improper sizing of dynamically-allocated buffers [62]. However, they could not fully replicate the experiment [51] as the datasets and scripts were not available anymore, and the original paper did not provide sufficient detail on how to re-implement the features extraction step. For these reasons, Riom et al. could not provide a faithful comparison. Yang et al. [52] considered the case of web vulnerabilities arising in Mozilla Firefox, and, using a large set of process and product metrics drawn by Kamei et al. [35], they provided a VPM that achieved high precision (over 90%), at the cost of having a very low recall score (below 15%) at the best possible configuration.

Our work and contribution. Table 1 summarizes and compares the works in the vulnerability prediction field, other than highlighting the main differences of our contribution. Our research aims at shedding lights on the capabilities of a large variety of machine learning models for just-in-time vulnerability detection. Hence, with respect to most of the papers discussed, our study has a different level of granularity and aims at assessing whether and how the promising research on machine learning for vulnerability detection can be applied at commit-level.

In particular, our study can be considered complementary with respect to previous works by Perl et al. [51] and Yang et al. [52] that targeted a commit-level granularity. First, we exploited multiple machine learning algorithms with the aim of providing a broader overview of how effective these techniques are for the just-in-time vulnerability detection, instead of employing only a single learner (e.g., SVM or Random Forest). Then, we employed a set of techniques to improve the model performance, such as removing features exhibiting multi-collinearity [39], balancing the dataset [40], and fine-tuning the model hyperparameters [41]. Such techniques were not always considered in the past when building VPMs, as also pointed by [32]. We also considered the role of textual metrics, which have been shown as highly relevant by Scandariato et al. [27]. In particular, we wanted to asses whether the raw use of the textual metrics actually provides an improvement in terms of predictive performance when considered with other features, as show by Theisen and Williams [31]. Finally, we targeted a different programming language, like Java, which has its own peculiarities and, more importantly, vulnerabilities. Indeed, a large part of the current body of knowledge covered types of weaknesses strictly tied to the programming language, e.g., the Buffer Overflow [62] vulnerability predominantly affecting C/C++ code.

On the basis of these considerations, the main contributions of our study pose an additional ground for software engineering researchers working on the identification of vulnerabilities, who can exploit our results to understand and build upon the current limitations and challenges connected to the application of machine learning-based vulnerability detectors at commit-level.

3. Research Methodology

In this section we provide a formulation of the design of our study according to the Goal-Question-Metric (GQM) paradigm [63]. In Section 3.1 we define the goal of our study and the consequent research question. Then, we describe the context of our empirical study, i.e., the projects we selected (Section 3.2), the procedures behind the automated extraction of vulnerability-contributing commits (Section 3.3), and the computation of software metrics (Section 3.4). All these data are required to build the dataset exploited by our machine learning pipeline, for which we provide a detailed description (Section 3.5). We conclude the section by presenting the evaluation methods we employed to answer our research question (Section 3.6).

3.1. Goal and Research Question

The *goal* of this empirical study was to investigate the performance of machine learning methods when employed for the task of just-in-time vulnerability detection, with the *purpose* of assessing their suitability in a pull-based development scenario. The *perspective* is both of practitioners and researchers: the former are interested in understanding whether and to what extent machine learning-based vulnerability detectors can be used during their daily activities; the latter are interested in evaluating strengths, weaknesses, and challenges for the use of machine learning for just-in-time vulnerability detection and that can be investigated further in future research.

We analyzed how well different machine learners can identify commits contributing to vulnerabilities. In this respect, we were inspired by previous research on vulnerability prediction [31] and assessed the impact of three families of software metrics on the performance of different machine learning algorithms. We asked:

RQ. *How well do machine learning algorithms perform when employed in the context of just-in-time vulnerability detection?*

We set up a machine learning pipeline that implements well-established guidelines for to the creation of unbiased supervised learning techniques [64, 38]. As further explained in the next sections, we considered and mitigated common pitfalls related to feature selection, hyperparameter configuration, data balancing, selection of performance metrics, and statistical tests. When designing and reporting our study, we adopted the guidelines by Wohlin et al. [65] and followed the *ACM/SIGSOFT Empirical Standards* [66].[5]

Table 2: Summary of projects considered in this study. These statistics are related to the period before 8th Match 2021.

Project	#Commit	LOC	#Sample Commits	#VCCs
Conversation	5,810	16,035	1,000	10
Candlepin	8,646	30,875	300	3
Hawtio	8,354	3,705	1,200	12
Jboss-Negotiation	299	505	191	2
Jenkins	25,867	29,080	4,400	44
Jolokia	1,573	3,685	1,100	11
Junrar	221	1,325	100	1
Litemall	990	3,500	100	1
Struts1-Forever	4,526	4,025	600	6
	56,286	**92,735**	**8,991**	**90**

3.2. Context of the Study

The *context* of the empirical study was composed of nine Java projects, whose main characteristics are reported in Table 2. These projects account for a total of 56,286 commits but, due to computational reasons, we randomly sampled 8,991 of them (16% of the total commits). When selecting the commits to analyze, we made sure not to discard commits containing vulnerabilities, whose collection is explained later in Section 3.3.

More in general, we considered all the Java projects having public software vulnerability data stored on the *National Vulnerability Database* (NVD). This database was originally created by the U.S. NIST Computer Security Division [67] with the aim of collecting and disclosing known vulnerabilities affecting software systems and their causes. It includes a comprehensive set of publicly known vulnerabilities: each of them is described through CVE (Common Vulnerabilities and Exposure [68]) records and is enriched with additional pieces of information such as external references, severity (computed using the Common Vulnerability Scoring System - CVSS), the related weakness type (Common Weakness Enumeration - CWE), and the known affected software configurations (Common Platform Enumerations - CPEs). NVD aggregates information from multiple data sources and is widely considered a reliable data source [69, 70, 71]. As a matter of fact, vulnerability reports must fulfill a well-defined set of requirements[6] before being added into NVD. As an example, vendors requesting for the creation of a CVE record have to provide a prose description of the issue, containing enough information for readers to understand which are the known products affected (e.g., application, operating system, or hardware). Such a description has to be supported by at least one accessible reference, e.g., a public mailing list. Moreover, a CVE describes one and only one independently fixable vulnerability, meaning that each record describes a single instance of an issue concerning a violation to the security policy of a product. This makes us confident enough about the validity and quality of the information contained in NVD.

Our focus on Java was motivated by the fact that previous research on software vulnerabilities did not extensively targeted this programming language (see Table 1): as such, our study can be considered as the first investigation of the capabilities of just-in-time detection approaches for the identification of known Java vulnerabilities. In addition, our choice was based on the availability of metrics that could characterize different aspects of Java source code, as well as the tools that could automate the data collection procedures.

3.3. Collecting Vulnerability-Contributing Commits

When collecting software vulnerabilities, we mined data exploiting CVE-Search,[7] an open-source tool that imports the entire set of CVE records from NVD into a MongoDB database for easier search and processing. We performed some additional filtering steps with the aim of removing incomplete/incorrect data that might have biased our conclusions: (1) we discarded CVEs that reported commits pointing to more than one GitHub repository, since we could not establish which project was involved in the first place; (2) we filtered out vulnerabilities whose fixes were marked as `merge` commits, as these do not apply any modification in the project history but simply incorporate the changes from a branch into another, i.e., we could not consider them as actual patches since we were interested in getting precise information about the time when fixes were added into the history rather than the time when they were sent into the main branch. After these filtering, we ended up with a total of 27 vulnerabilities (CVEs) of 12 different types (CWEs).

Afterwards, we implemented a mining procedure based leveraging the well-known SZZ algorithm [72] to fetch the vulnerability-contributing commits (VCCs) [73], i.e., the commits that are likely to have contributed to the introduction of a vulnerability. To this purpose, we started from vulnerability-fixing commits that we mined from NVD. Specifically, for each file f_i touched by the fixing commit c_{fix}, our algorithm runs the `git-diff` command to extract the list of modified lines in f_i with respect to the previous commit c_{fix-1}; then, it runs the `git-blame` command on the deleted lines in order to retrieve the commits where these were changed last. We consider these commits

[5] Given the nature of our study and the currently available empirical standards, we followed the *"General Standard"* and *"Data Science"* definitions and guidelines.

[6] `https://www.cve.org/ResourcesSupport/AllResources/CNARules#section_8-1_cve_record_information_requirements`

[7] https://github.com/cve-search/cve-search

as VCCs of the vulnerability fixed in c_{fix}. As a result, for each vulnerability we obtain a set of VCCs as more than one commit might contribute to its introduction.

To improve the precision of this procedure, we applied some additional adjustments to reduce the risk of catching false positive VCCs. We excluded files from c_{fix} that were (1) non-source JAVA files, (2) test classes, (3) build files, (4) documentation and blob resources (the entire list of blacklisted files is available in our online appendix[4]). We also filtered out the VCCs that appeared as `merge` commits, as they do not report any *actual* modifications to the project's history—indeed, we were interested in the moments in which the patches were added in the history for the first time, not when they were merged into the main branch [74]. Finally, we managed the cases where the fixing commit c_{fix} consisted only of added lines. In these situations, there are no lines to blame and we assumed that the files involved in the commit were born vulnerable: as such, we marked the commits that introduced the files as vulnerable. Overall, we managed to obtain a total of 90 distinct VCCs among the nine projects—a detailed list reporting these commits is available in our online appendix.[4] Whether or not a commit contributes to a vulnerability represents the *dependent variable* of the models built, i.e., the information that we aimed at predicting using machine learning techniques.

3.4. Collecting Software Metrics

Once we had collected vulnerability data, we focused on the *independent variables*. In this respect, we exploited three families of metrics that were investigated in previous studies on software vulnerability detection: process, product, and textual features. The detail of each of these metrics, along with the description and the rational behind their usage, is described in Table 3.

With respect to process metrics, we considered different aspects previously treated in vulnerability research [55, 29, 28, 51, 58] and able to characterize the change history of the projects, like the churn metrics (concerning added and deleted lines, methods, conditions, method calls, and assignments), the extent of contribution made by the committing author (i.e., the developer implementing the change), the number of files involved in the commit, the scattering of the changes, the number of previous changes and author of the files, etc. To compute these metrics, we developed our own tool, available in our online appendix.[4] It is worth point out that most of these metrics concerns metadata directly extracted from the commit metadata—e.g., the number of days between the commit date and the project creation date—while two of them, namely *Mean Days Since Creation* and *Mean of Past Changes*, were obtained by analyzing the `git` metadata related to each file involved in the commit. For these metrics, we aggregated the values obtained from each valid file (with the same filters used in Section 3.3) using the mean operator to bring them at commit-level, enabling their use as predictors for the machine learning models.

As for product metrics, we took into account the Chidamber & Kemerer suite [53], a set of well-known Object-Oriented metrics able to quantify different structural properties of the source code, such as cohesion and coupling. Similarly, to process metrics, we exploited an ad-hoc tool, available in our online appendix,[4] able to extract structural metrics from a given parsable JAVA files. To reach our goals, we run it against all the JAVA files involved in the commits to extract the traditional set of CK metrics (listed in Table 3); afterwards, we computed the mean of the metric values to bring them at commit level, similarly to what was done by Yang et al. [52] for the SLOC metric.

Finally, we extracted the textual features experimented by Scandariato et al. [27]. For each commit, we selected the valid files (which underwent to the usual filters described in Section 3.3) so that we could make our document corpus. Then, we extracted its *bag-of-words* [56, 57], which is a compact representation of the documents in the corpus through the number of occurrences of the words (a.k.a. terms) appearing in the entire corpus (which constitute the *vocabulary*). Namely, a file is represented as a vector of M integers, each representing the counting of the M words appearing in the vocabulary. At this point, the bags-of-words of the files involved in a commit were summed together, so that any commit could have its own bag-of-words made of the total number of times each words appeared in the modified valid files only. We treated each term as an independent variable for our models. To remove any noise that could damage the models performance [75, 76], we filtered out the high-frequency words—removing the ones appearing in more than 80% documents, as they add poor information to the text; in addition, we also dropped low-frequency words, appearing in less than 5% documents, to reduce the dimensionality of the feature space, which was shown to improve the training process [77, 78]. All in all, we ended up with 1,318 distinct tokens, each encoded as a numeric feature. In addition, following the approach adopted by Perl et al. [51], we extracted the bag-of-words of the sole commits' patches to count the terms involved in the actual change, without considering the unaffected code areas. Specifically, for each commit we obtained the bag-of-words of the added lines only, and the bag-of-words of the deleted lines sharing the same vocabulary. Then, we computed the absolute difference between the two vectors, so that we could obtain the number of times each term was involved, either in an addition or deletion, in the actual patch. Also in this case we filtered out high- and low-frequency words using the same filters used for the entire files. In this case, we ended up with 128 distinct tokens, encoded as 128 integer features. It is worth remarking that we did not compute the overlap between these 128 terms and the 1,318 extracted in the previous step, as they originate from two different corpuses (i.e., files and patches). Thus, we considered a total of 1,446 tokens. These two approaches were implemented in our own scripts, which relied on SCIKIT-

LEARN's `CountVectorizer` class,[8] and made it publicly available into our appendix.[4]

3.5. Setting the Machine Learning Methods

After having collected dependent and independent variables to be used, we configured the machine learning models to detect vulnerable commits. The design of our machine learning pipeline is described hereafter.

3.5.1. Design of the models

As we had collected different families of metrics, we could experiment with various models. We first devised three supervised techniques that relied, individually, on *product*, *process*, and *textual* metrics to predict the proneness of the commits to be vulnerable: in this way, we could assess the contribution given by each metrics set. Afterwards, we started combining them by adopting a stepwise method: we created models based on *product+process*, *product+textual*, and *process+textual* features. Finally, we also considered the model using all the features together. As a consequence, we designed and experimented with seven different combinations of features.

3.5.2. Selection of the classifier

We treated the problem as a binary classification task: determining whether a commit contributed to a vulnerability or not. As discussed in Section 2, the related literature did not pose conclusive results on the machine learning algorithms that are more suitable for the classification of software vulnerabilities. For this reason, we experimented with the following eight learning algorithms:

SUPPORT VECTOR MACHINE (SVM) [92]. This is a statistical model that constructs the best hyper-plane out of the infinite possibilities in a N-dimensional space—with N being the number of features. The best hyper-plane is capable of distinctly separate the data points, having the maximum margin (namely the largest distance to the nearest training data points of any class).

KNEARESTNEIGHBORS (KNN) [93]. This a non-parametric technique that classifies the samples using the dataset alone (i.e., without building a model). The classification is made as a majority vote, i.e., based on the class of the majority of its k nearest neighbors data points.

DECISION TREE [94]. This is a classifier with a tree-like structure, characterized by multiple *nodes* and *leaf*. The nodes are linked through branches, representing a test. The output is given by the decision path taken. The decision tree is structured as an if-then-else diagram: given an input variable (root node), it leads to multiple sub nodes through branches. The process is iterated until the output (leaves) is reached.

RANDOM FOREST [95]. This is an ensemble technique that helps to overcome the overfitting issues of the decision tree. Ensemble means that this model uses a set of *weak* classifiers (decision trees in this case) to solve the assigned problem. Each individual tree is generated using a random subset of samples in the dataset. To reduce the correlation between the individual trees, the splitting point is chosen using a random subset of the dataset, *without replacement*. Using this method, a RANDOM FOREST is able to better generalize the data and reduce the overfitting problem faced by other classifiers.

EXTREMELY RANDOMIZED TREES [96]. ExtraTrees adds a further randomization to the RANDOM FOREST, as each node of the weak classifiers is split randomly. This means that instead of relying to specific metrics for choosing the optimal splitting point, this model randomly generates a series of splits and choose the one which gives the best result. This characteristic allows the model to be less computationally expensive compared to the others, while maintaining high generalization capabilities.

ADABOOST [97]. This is an ensemble model based on *boosting* [98], in which each individual tree is trained in a *sequential* fashion. Initially, a single decision tree is created and the same weight is assigned to all samples in the training set. Progressively, the weights are increased for the misclassified samples and another tree is generated. The whole process continues until a predefined number of trees has been generated or the accuracy of the model cannot be improved anymore. With respect to the other ensemble models, ADABOOST is less prone to overfitting.

GRADIENT BOOSTING [99]. As ADABOOST, it uses an ensemble of individual trees which are generated sequentially. A tree is generated after each iteration to minimize a differential loss function. The process stops when the predefined number of trees has been created or when the loss function no longer improves.

XGBOOST [100]. An improved implementation of GRADIENT BOOSTING algorithm, allowing faster computation and parallelization.

The choice of focusing on these classifiers was driven by our willingness to investigate the classification performance of a large variety of algorithms, including ensemble methods. It is worth remarking that in our research we were interested in benchmarking *narrow* artificial intelligence techniques [101]: the evaluation of other approaches belonging to the category of *strong* artificial intelligence, e.g., deep learning, is part of our future research agenda.

3.5.3. Preprocessing steps

As recommended in literature [38], we performed a number of steps aimed at building a machine learning pipeline that could avoid bias in the interpretation of the results. In the first place, we applied a *feature selection* in order to avoid multi-collinearity [39]. This step was required to remove correlated metrics that provide the

[8] `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html`

Table 3: List of metrics extracted from each commit in the dataset, used as independent variables (features) for the machine learners. The table reports a description, the rationale behind their selection, and related works in which they have been used for VPMs.

Name	Description	Rationale	VPMs
Process Metrics			
Added Lines	The number of lines added in the commit.	A high amount of added lines indicates a large commit, which has a higher risk of introducing defects [79, 80, 81] or vulnerabilities [29, 73, 28].	[29, 48, 51, 52]
Deleted Lines	The number of lines removed in the commit.	Same as *Added Lines*.	[29, 48, 51, 52]
Added Methods	The number of new functions/methods added in the commit.	New functions or methods may add new security check or increase the attack surface [82, 52].	[51, 52]
Deleted Methods	The number of removed functions/methods in the commit.	Deleting security-critical functions or methods may remove security checks or reduce the attack surface [82, 52].	[51, 52]
Modified Methods	The number of changed functions/methods in the commit.	The removal of security-critical functions or methods may modify the security profile [82, 52].	[51, 52]
Added Conditions	The number of added conditional expressions in the commit.	Same as *Added Methods*.	[52, 61]
Removed Conditions	The number of removed conditional expressions in the commit.	Same as *Removed Methods*.	[52, 61]
Added Method Calls	The number of added function or method call in the commit.	Same as *Added Methods*.	[52, 61]
Removed Method Calls	The number of removed function or method call in the commit.	Same as *Removed Methods*.	[52, 61]
Added Assignments	The number of assignments added in the commit.	Adding new assignments may improve or drop security constraints [82, 52].	[52, 61]
Removed Assignments	The number of assignments removed in the commit.	Same as *Added Assignments*.	[52, 61]
Mean Days Since Creation	The mean number of days elapsed from the creation dates of each modified file to the commit date.	The "age" of each file could be correlated with the presence (or absence [81]) of vulnerabilities.	N/A
Mean of Past Changes	The mean number of previous changes (i.e., commits) of each touched file.	A file that was changed many times is more prone to defects [80] and vulnerabilities [29, 28, 51].	[29, 28, 51, 52]
Past Different Authors	The size of the set of distinct authors that touched the files modified in the commit.	A file touched by many different developers is more prone to defects [80] and vulnerabilities [29, 28, 51, 79, 73].	[29, 28, 51, 52]
Author Past Contributions	The number of commits done by the author before the commit.	Inexpert developers may involuntarily contribute to vulnerabilities [73].	[52]
Author Past Contributions Ratio	*Author Past Contributions* divided by the total number of commits made to the entire project.	Same as *Author Past Contributions*.	[51]
Author 30-days Past Contributions	The number of commits done by the author within 30 days before the commit date.	Same as *Author Past Contributions*.	N/A
Author 30-days Past Contributions Ratio	*Author 30-days Past Contributions* divided by the total number of commits made 30 days before the commit date.	Same as *Author Past Contributions*.	N/A
Author Workload	The amount of work that an author has invested within a 30-days time window. Given a commit x performed on date d, we considered the distribution of commits done by the developers involves within 30 days before d, scaled to [0..1] range, and assigned its percentile value [83].	A developer with high workload could implement poor quality code [83], defects [84] or vulnerabilities [29].	N/A
Days After Creation	The number of days elapsed from the project's repository creation (i.e., the first commit) date to the commit date.	The "age" of the repository has an impact on the general code quality [83] and the introduction of errors [85].	N/A
Fix	Whether or not the commit had the goal to fix an issue or a defect. This is done by looking at specific keywords in the commit message (reported in our online appendix).	Fix commits may cause collateral damages of introducing new bugs [35] or vulnerabilities [86].	[52]
Touched Files	The number of files modified in the commit, excluding the irrelevant ones (test, documentation, build, and blob files).	A commit touching many files lacks of cohesion and may have a higher risk of introducing defects [51, 87, 52].	[52, 61]
Entropy of Changes	Distribution of changes across each modified file, measured using the Normalized Static Entropy, as used by Kamei et al. [35].	A high entropy indicates a highly-fragmented commit, i.e., scattered changes touching many files, which indicates a highly-complex commit [88, 35, 89].	[52]
Number of Hunks	The number of continuous blocks of changes in the commit diff.	Similar to *Entropy of Changes*.	[51]
Product Metrics			
LOC	Lines of Code, counting both source and comment lines.	Large files tend to have higher risk of becoming vulnerable [90, 29, 79, 51, 30].	[29, 46, 48, 50]
SLOC	Source Lines of Code, i.e., LOC without comment and documentation lines.	Same as *LOC*.	[46, 52]
WMC	Weighted Methods per Class, i.e., the sum of the complexities (i.e., McCabe's Cyclomatic Complexity) of all the methods in a class [53].	Complex code is difficult to maintain and test [28, 46, 54] and thus has higher chance of having vulnerabilities [28, 46, 29].	[29, 46, 50]
CBO	Coupling Between Object, i.e., the number of dependencies a class has with other classes [53].	Highly coupled code makes input from external sources harder to trace [28], and has positive correlation with vulnerabilities [46].	[29, 46]
RFC	Response For a Class, i.e., the number of methods (including inherited) that can potentially be called by other classes [53].	Same as *CBO*.	[46, 28, 50]
DIT	Depth of Inheritance, i.e., the depth of the class within its inheritance tree [53].	A deep class is likely to have a larger number of inherited methods, making it more complex to predict its behavior as it is affected by many ancestor classes [46].	[29, 46]
NOC	Number of Children, i.e., the number of direct sub-classes [53].	Changing a class with many incoming dependencies may introduce defects [46].	[29, 46]
LCOM1	Lack of Cohesion of Methods version 1, i.e., the number of pairs of methods not sharing all the fields they access to [53].	Poor cohesive code has been shown to be positively correlated with vulnerabilities [46].	[46]
LCOM2	Lack of Cohesion of Methods version 2, i.e., the percentage of methods not accessing a specific attribute averaged over all attributes in the class [91].	Same as *LCOM1*.	[46]
Text Metrics			
Files Term(s) Frequency	The counting of each word that appears in the full text of the modified Java files.	Term frequency has been shown to improve the prediction power if considered with other metrics [50, 31].	[27, 49, 50]
Patches Term(s) Frequency	The number of times in which the words appearing in the patches involving Java files were changed (added or removed).	Same as *Files Term(s) Frequency*.	[51]

machine learners with the same (or similar) information and that might cause them to not being able to derive the correct explanatory meaning of the features. In this respect, we exploited the Variable Inflation Factor (VIF) method [39]: for each independent variable and for each experimented model, the `vif` function measures how much the variance of the model increases because of collinearity. The features having a `vif` coefficient higher than 5 were removed; the process was repeated until the point where all the features had coefficients lower than the threshold. Afterwards, we considered the problem of hyper-parameter configuration. In particular, we run the RANDOM SEARCH algorithm [41], which performs a randomized search of the hyper-parameter space with the aim of identifying the optimal hyper-parameter values to use for the classification task. Bergstra et al. [41] proved that this search algorithm is able to reach, using less computational resources, the same—or even better—hyper-parameter configuration as an exhaustive search, e.g., GRID SEARCH.

3.6. Evaluating the Machine Learning Methods

Our empirical investigation led to the training and validation of a total of 56 different models, coming from the combination of the eight machine learning algorithms (Section 3.5.2) and the seven features combinations (Section 3.5.1). The results of the comparison of these models are reported in Section 4. After setting the machine learners, we defined the data analysis procedure to address our research question.

3.6.1. Training and Validation strategy

To assess the capabilities of the considered models, we had to define a training and validation strategy. We took into account the imbalance of the dataset: as previously shown (see Table 2), each project has around 1% of vulnerable commits. As such, we applied the *Synthetic Minority Oversampling Technique* (SMOTE) [102]: for each project, this technique generates artificial samples of the minority class (i.e., vulnerable commits in our case) in order to rebalance the classes. Unfortunately, we found that the technique could not be applied on all the considered projects. In particular, SMOTE requires the presence of at least two samples of the minority class; otherwise, it does not have enough data to oversample the dataset. In two projects, i.e., JUNRAR and LITEMALL, only one commit was labelled vulnerable and it was not possible to apply the balancing approach. This problem influenced our training procedures, as we could not effectively train machine learners using a *within-project* strategy.

Hence, we went for a *cross-project* training. This means that we aggregated data coming from *n-1* projects, balance the training set, and then verify the performance of the models on the remaining project. More specifically, we adopted a *Leave One Group Out* (LOGO) validation strategy, which divides the entire dataset into folds, each containing all the commits of a single project for a total of 9 folds. The validation consisted of 9 iterations, each using 8 folds to build the training set, and the remaining one for the test set. As a consequence, each project was used in *n-1* training sessions, and only once for the testing.

3.6.2. Detection performance measures

For each fold experimented during the validation, we assessed the machine learning models capabilities using a number of performance measures. First, we computed *precision* and *recall*. However, as suggested by Powers [103], these two measures present some biases as they are mainly focused on positive examples (i.e., vulnerable commits in our context) and predictions, so they do not capture any information about the rates and kind of errors made. The contingency matrix (a.k.a. confusion matrix), and the related *F-measure* overcome this issue. Moreover, we computed the *Matthews Correlation Coefficient* (MCC) [104] to understand possible disagreement between actual values and predictions—the coefficient involves all the four quadrants of the contingency matrix. In addition, from the contingency matrix we retrieved the measure of *true negative rate* (TNR), which measures the percentage of negative sample correctly categorized as negative, *false positive rate* (FPR) which measures the percentage of negative sample misclassified as positive, and *false negative rate* (FNR), measuring the percentage of positive samples misclassified as negative. The measure of *true positive rate* is left out as equivalent to the recall. Finally, we computed the *Receiver Operating Characteristic* (ROC) curve, and the related *Area Under the Curve* (AUC-ROC). This measure gave us the probability of ranking a randomly chosen positive instance higher than a randomly chosen negative one.

3.6.3. Statistical Analysis

The final step of our methodology consisted of the application of statistical tests to verify whether the differences in the performance achieved by the various experimented models were statistically significant. Such an analysis was useful to assess the existence of metrics and/or classifiers that were more suitable for the problem of just-in-time vulnerability detection. Since the data are not normally distributed, we exploited the Friedman Test with the Nemenyi post-hoc test [105] on all the machine learning models. This is a post-hoc test that identifies the groups of data that differ after a statistical test of multiple comparisons has rejected the null hypothesis (the groups are similar), making a pair-wise performance. We selected this test because it is robust to multiple comparisons - which is our case since we had to compare multiple models on multiple features - and does not require the underlying distribution to be normally distributed. To conduct the statistical analysis, we used the `Nemenyi` package for PHYTON.[9]

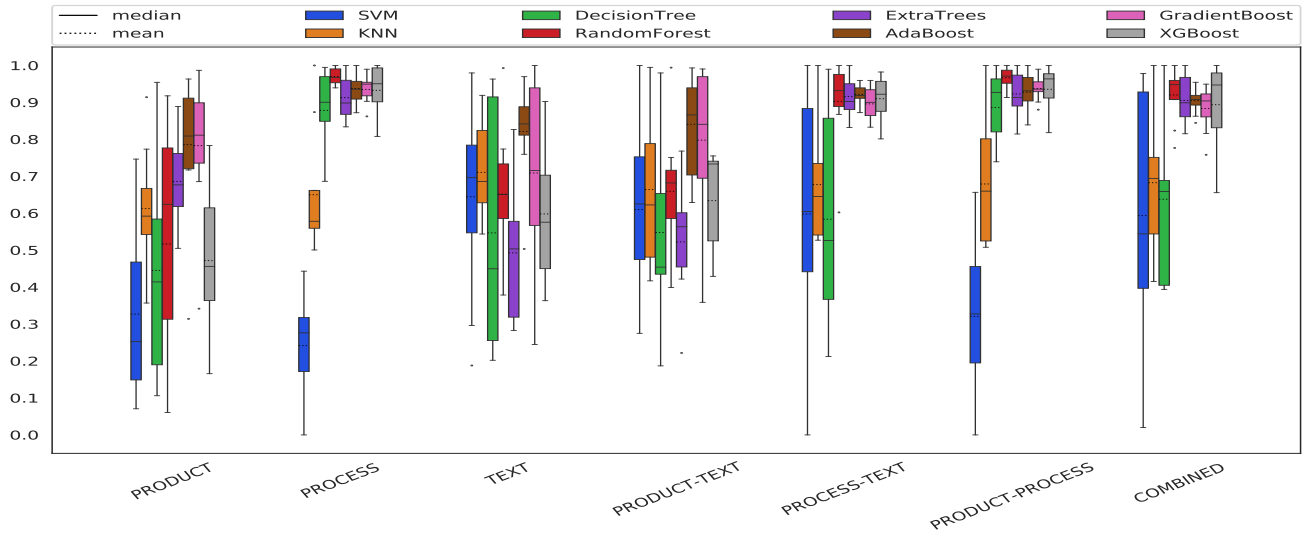


Figure 1: The AUC-ROC scores obtained during the LOGO validation of the 56 models, grouped by the seven features combinations.

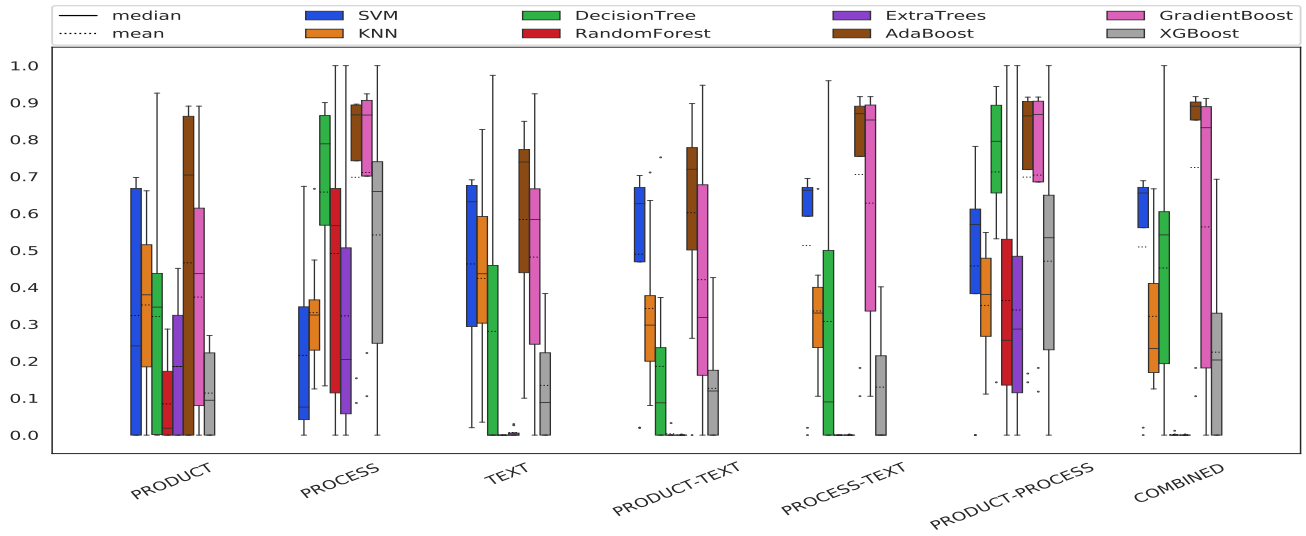


Figure 2: The F-measure scores obtained during the LOGO validation of the 56 models, grouped by the seven features combinations.

4. Empirical Study Results

Figure 1 and Figure 2 depict the box plots reporting the distribution of AUC-ROC and F-measure values obtained during the LOGO validation of the 56 machine learning models on the considered dataset. In both figures, each color indicates the model produced by the selected learning algorithms (Section 3.5); the box plots were also grouped by the seven different combinations of features. For the sake of readability and comprehensibility, we only report in detail the results of two of the seven performance metrics described in Section 3.6; however, the complete results are included in the online appendix.[4]

Considering the AUC-ROC distributions (Figure 1) the ensemble methods (Random Forest, Extra Trees, AdaBoost, Gradient Boosting, and XGBoost) generally performed better than the three basic classifiers (SVM, KNN and Decision Tree) over all the seven combinations of features. Among all the feature sets, the *product* group alone (label *"PRODUCT"*) caused the models to obtain the worst AUC-ROC scores. Something similar, though with lesser extent, happened for the *textual* metrics (label *"PRODUCT"*). The combination of these two groups (label *"PRODUCT-TEXT"*) did not yield any relevant positive effect, i.e., the addition of *product* metrics does not provide substantial changes in terms of AUC-ROC. Moreover, the sole presence of *product* and/or *textual* metrics did not highlight any relevant difference between basic classifiers and ensemble models, i.e., they are comparable in terms of AUC-ROC. Although the ensemble models still show better performance, the SVM and

[9] `https://scikit-posthocs.readthedocs.io/en/latest/`.

KNN models are the only ones that greatly benefit from the presence of *textual* metrics. This phenomenon becomes even more evident when moving from the *product+process* group to the *combined* one. The most interesting results occurred when *process* metrics were involved (all the groups having the *"PROCESS"* label in Figure 1). On the one hand, these metrics further increased the differences among the AUC-ROC distributions—e.g., the large gap between the box plots of AdaBoost and SVM. On the other hand, almost all the models—with the notable exception of the SVMs—received a general improvement. What is more, the ensemble models achieved the best AUC-ROC scores in *product+process* feature combination (label *"PRODUCT-PROCESS"*), hinting that the addition of *textual* metrics causes negative, though marginal, effects. Once again, SVM and KNN were not subject to these phenomena: their models did not receive any positive effect from the presence of *process* metrics. Indeed, similarly to the *product* metrics, they seem to be quite "insensitive" from the presence or absence of *process* metric when the *textual* metrics are already involved. This can be seen by comparing the set having the *textual* metrics alone (label *"TEXT"*) with the ones including them (*"PRODUCT-TEXT"*, *"PROCESS-TEXT"*, and *"COMBINED"*).

The F-measure trends (Figure 2) are largely different from the ones seen with the AUC-ROC. In the first place, not all the ensemble methods benefit from the presence of *process* metrics. Random Forest, Extra Trees and XGBoost classifiers scored even lower F-measures than the basic classifiers; this difference becomes even larger when *textual* metrics are added: their F-measures collapsed around 0. Differently, AdaBoost Gradient Boosting preserve the general behavior seen with the AUC-ROC: adding *process* metrics is always beneficial, i.e., the inter-quartile ranges shrunk, while the mean and median values increased. To a far lesser extent, these two models suffer from the presence of *textual* metrics, as they may slightly worsen their performance. As an example, XGBoost dropped for about 0.1 point in F-measure when *textual* metrics were added to *product* and *process* models (i.e., from *"PRODUCT-PROCESS"* to *"COMBINED"*). In any case, the sole presence of *process* metrics is enough to achieve acceptable performance.

Finding #1

The majority of the models benefit—in terms of both AUC-ROC and F-measure scores—from the presence of *process* metrics in the set of predictors. SVM- and KNN-trained classifiers give the best of themselves when *textual* metrics are involved, while the opposite, to varying degrees, occurs for the other models, especially in terms of F-measure—which is much more susceptible than AUC-ROC.

From the point of view of the machine learning algorithms, the Decision Tree provides the most unstable models, highly influenced by the set of predictors used, i.e., they received the largest drop in terms of both AUC-ROC and F-measure when *textual* metrics were added. This could be explained by the fact that decision trees are particularly sensible to noise in the training data, and cannot properly generalize. This effect is more obvious in the case of a high dimensional feature space—i.e., the one created when all the tokens from the two *bag-of-words* built are added—or highly imbalanced data—which is true in this context since the number of vulnerable instances is far lower than the number of "safe" instances. Such a limitation is partially solved by using ensemble methods. Conversely, the classifiers trained using SVM and KNN are the only models positively influenced by the presence of *textual* metrics. In particular, KNNs resulted to produce the most stable models, being the least influenced by other predictors that are not part of the *textual* group, and achieving very similar scores in most combinations of predictors. Between the two, KNN outperformed SVM in terms of AUC-ROC, but it scored very low performance in terms F-measure. Nevertheless, both algorithms did not manage to train models with high scores, making them unsuitable in the context we considered.

Random Forest and Extra Trees, despite having a similar learning mechanisms, obtained quite different distributions: they both scored the worst possible F-measures, being around 0 in most cases, even lower than a traditional Decision Tree. They draw benefit from the inclusion of *process* metrics, but they are too much negatively influenced by the tokens of the *bag-of-words*. Curiously enough, they still managed to reach very high AUC-ROC scores, sometimes even outperforming all the other learners. Such contrasting AUC-ROC and F-measure values implies that there is a possibility to improve the predictive capabilities of these models by tuning the decision threshold: instead of keeping it to the default value 0.5, change it accordingly to the specific needs, finding the best trade-off between the recall and the false positive rate, which also have an impact to the F-measure.

The scenario is thoroughly different for boosting-based models: AdaBoost, followed by Gradient Boosting, outperformed the other learners on all fronts. This result was somehow expected since both these models build sequential *shallow* weak classifier, usually a single split decision tree, which are less prone to overfit compared to the *deep* weak classifiers used by other ensemble models like Random Forest. Moreover, the aggregation of the prediction is weighted in the case of AdaBoost and Gradient Boosting, hence the individual weak classifiers who performed better have a higher weight compared to those who performed poorly. In the other ensemble models, the prediction of each weak classifier carries the same weight. Oddly enough, XGBoost, despite being a boosting-based model, was very far from the performance of AdaBoost and Gradient Boosting, and more similar to Random Forest and Extra Trees.

Finding #2

Boosting-based classifiers, especially AdaBoost and Gradient Boosting, achieved the best overall results. The other ensemble models scored far worse F-measures, even lower than basic classifier. RandomForest and ExtraTrees, however, obtained very high AUC-ROC scores, hinting the possibility to improve their predictive capabilities by properly tuning the decision threshold. SVMs and KNNs are the only models to benefit from *textual* metrics, and generally ignore the effect of other features.

To assess whether the distributions of the performance metrics were statistically different when considering different combinations of predictors, we run the post hoc Nemenyi rank test [105] on all the machine learning models. For the sake of readability, in this paper we only report and describe the results for AdaBoost, i.e., the algorithm that provided the best results over all the seven combinations of features. For consistency, we show the p-values of the Nemenyi rank test computed on the distribution of AUC-ROC and F-measure values by the means of heatmaps (Figures 3a and 3b). In addition, we report the statistical results (in terms of AUC-ROC and F-measure) of the eight experimented machine learners trained using the *product+process* features set, i.e., the best combination according to our results (Figures 4a and 4b). The complete results are reported in our online appendix.[4]

Figure 3a shows statistically significant differences (depicted in dark violet) in AUC-ROC values between the models built using the *product* metrics alone (label *"PRODUCT"* label) and both (1) those built using *process* metrics (label *"PRODUCT"*), and (2) the ones trained using only the *textual* metrics (label *"TEXT"*). This confirms the large positive effect that *process* metrics have on the AUC-ROC measure on AdaBoost-trained models. Between the *product* and *textual* groups there are no statistically significant differences, implying that there is no sufficient evidence to establish which provides higher predictive capabilities. On a similar note, Figure 3b shows the presence of statistically significance differences between the *combined* group (label *"COMBINED"* and the groups involving either *product* or *textual* metrics (labels *"PRODUCT"*, *"TEXT"*, and *"PRODUCT-TEXT"*) in terms of F-measure. This is a further evidence on the contribution provided by the *process* metrics.

Focusing on the *process+product* combination, which provided the best models overall, Figure 4a better highlights the comparable performance obtained by the ensemble methods which significantly differs from the ones obtained by SVM and KNN—which did not benefit from the *process* metrics, but, rather, from *textual* ones. Figure 4b provides a different view of what could be seen from the box plots (Figure 2): AdaBoost and Gradient Boosting far greatly surpassed the F-measures scored by RandomForest, ExtraTrees, and DecisionTree models. Surprisingly, the DecisionTrees were able to significantly surpass the performance of RandomForest and ExtraTrees. This does not immediately implies that decision trees are better than the related ensemble methods. As a matter of fact, RandomForest and ExtraTrees still scored higher AUC-ROC, suggesting the need to fine tune the decision threshold to achieve better predictive capabilities, instead of relaying on the default one (which could also be the best choice in certain cases). This aspect, however, deserves further investigation.

Finding #3

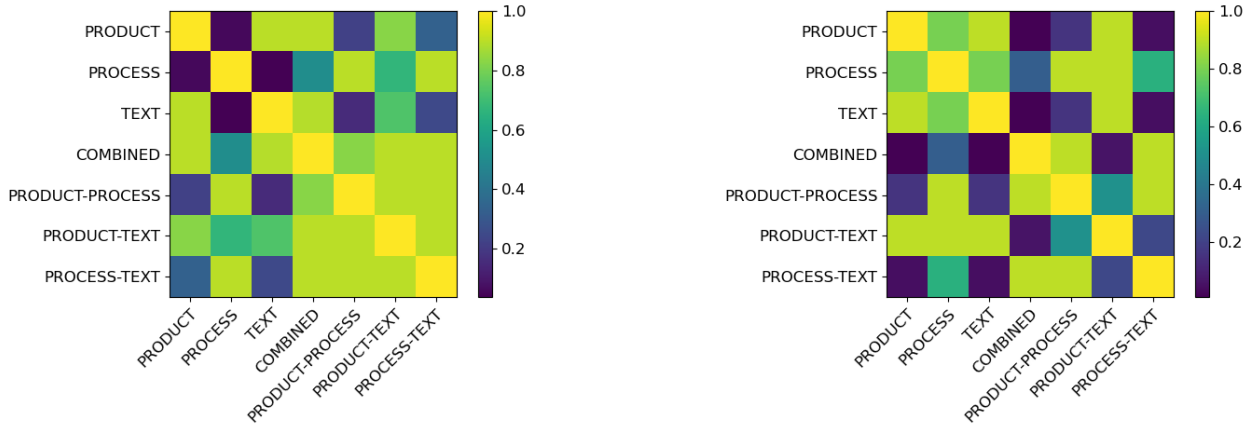
Significance tests confirm the findings discovered during the qualitative analysis of the distributions by the means of box plots: the adoption of *process* metrics to AdaBoost models provides improvements in terms of both AUC-ROC and F-measure. More in general, the boosting-based algorithms are better than other classifiers, while non-boosting ensemble methods still need further investigation on how to improve their capabilities by tuning their decision thresholds.

5. Discussion and Implications

The results achieved in our empirical study revealed a number of insights that may lead to concrete implications for the software engineering research community, and that we further discuss hereafter.

Comparison with other just-in-time VPMs. Our analyses revealed a number of insights that could be related to the ones discovered by Perl et al. [51] and Yang et al. [52], i.e., the closest studies to our work and that represent the current state-of-the-art in just-in-time vulnerability prediction modeling. Similarly to what Riom et al. experienced [61], we could not provide a precise comparison with the VPMs described in [51] and [52], as the original papers point to appendices that no longer exist, preventing us to access to the raw results they achieved. Moreover, the description of the metrics extraction provided in those papers do not report implementation details, making the reproduction even harder. For all these reasons, we only compared our findings with the ones reported in [51] and [52], leaving out any detailed comments on the actual performance scores achieved by the models. In this respect, our goal was to find any possible point of agreement and/or disagreement between our contribution and the current state-of-the-art VPMs.

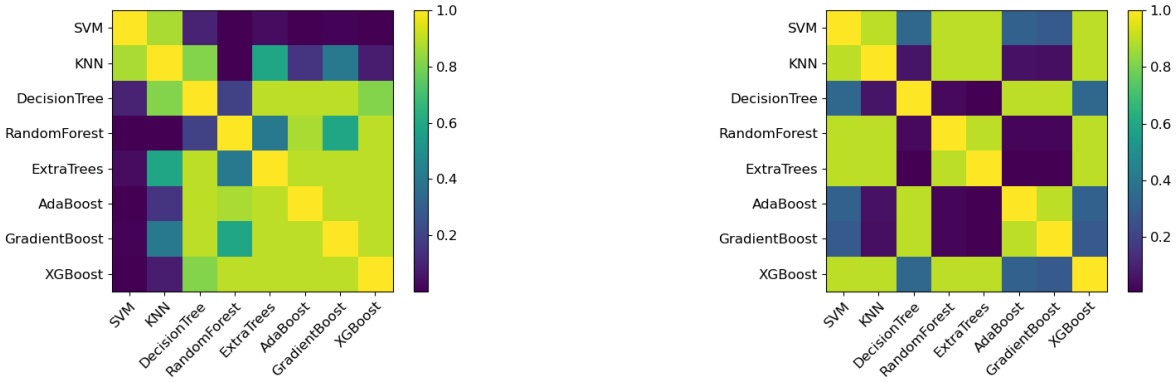
The performance of the two VPMs [51, 52] were reported in terms of precision, using the rationale that, in the context of predicting software vulnerabilities, a higher precision is preferable as the minimization of the false positive rates is instrumental for preventing developers to pointlessly inspect a large number of commits that will not contribute to the insertion of vulnerable code. Because of this, the authors compared their models with a



(a) AUC-ROC

(b) F-measure

Figure 3: Nemenyi test p-values obtained for comparing the eight AdaBoost models trained on the seven features combinations.



(a) AUC-ROC

(b) F-measure

Figure 4: Nemenyi test p-values obtained for comparing the models trained on *product+process* feature combination using the eight machine learning algorithms.

baseline static analysis tools, i.e., FlawFinder [106], to find whether they could outperform its detection capabilities at the same recall level (done by varying the decision threshold). In both studies, the models were able to achieve much higher precision, i.e., they largely reduced the amount of false positives discovered by FlawFinder. Yet, such a comparison is still limited, as it does assess the actual effectiveness of machine-learning models. As a matter of fact, under these configurations—i.e., when setting the decision threshold to have the same recall level as FlawFinder—the SVM built in [51] obtained an F-measure of 0.343, while the RandomForest used in [52] scored 0.198. Both these results indicate limited effectiveness. It is worth remarking that we could not compare these scores with ours as all the studies considered different contexts and feature sets, making any comparison unfair and leading to wrong conclusions. In any case, the F-measure, together with precision and recall, cannot be the sole measure to be taken into account, especially when working with imbalanced datasets. Indeed, other measures, such as AUC-ROC and MCC, are recommended to provide a better overview on the predictive capabilities of the models [107, 40]. To the best of our knowledge, our study is one of the first in JIT vulnerability prediction that does not consider the precision alone, and whose primary goal is not overcoming static analysis tools, but rather comparing many learning algorithms to find which provides the best models, as well as adopting critical preprocessing steps aimed at improving the training session.

Both [51] and [52] considered the use of *textual* metrics under the "code metrics" feature group. Specifically, Perl et al. [51], not only they run the bag-of-words on the patch content, but they also added the counting of the C/C++ language keywords (e.g., `if`, `goto`, etc.) in the same group; simlarly, Yang et al. [52] counted the C/C++ keywords appearing in the modified files of the commit. The behavior of our SVM models seems to be in line with the one by Perl et al. [51]: the *textual* metrics seems to be beneficial, as opposed to the *process* metrics (which they called GitHub *meta-data*). This may be explained by the fact that SVMs are able to perform well even with large and sparse feature space, i.e., when considering words counting [78]. On the other hand, the RandomForest

in [52] obtained the worst performance when involving the *textual* metrics, the same encountered with our RandomForests. In both studies the authors managed to obtained the best performance when considering all metrics together, confirming the results observed in [31] at the file granularity level. Our SVMs did not experience this effect, as the best model is obtained when only *textual* features were considered, while our RandomForests confirmed this effect only for the AUC-ROC scores, as the addition of *textual* metrics dropped the F-measure close to 0. This hints the need of better data pre-processing activities tailored on the requirements of each learning algorithm.

JIT vulnerability detection: Are we there yet? In the title of the paper, we pose this question. According to our results, the answer is: *"No"*. The accuracy of the existing vulnerability prediction models is not enough to make developers aware of possible vulnerabilities when committing new changes onto a repository. Our study identifies a number of open issues and challenges that the research community should further consider and on which we elaborate more in the remainder of the section. From the predictive power of the features to the machine learning pipelines configured for the prediction exercise, the currently available solutions cannot provide a *just-in-time* feedback to developers. From a practical perspective, our results indicate the lack of techniques that can analyze the changes done within a commit and detect possible inconsistencies inducing vulnerabilities. As such, developers must still rely on longer-term predictions that analyze entire releases to identify vulnerable files. This represents a threat to the usability and usefulness of the available approaches, as indicated by previous work [35]. Hence, our work points out the need for further research on the matter and that should be devoted to all components of the machine learning pipelines.

The existing metrics are not enough. One of the key outcomes of our research is the inability of current metrics to characterize vulnerable commits in an effective and consistent manner. Indeed, despite having considered most of the metrics exploited in previous work on VPMs, in many cases the performance achieved in terms of F-measure are low. This is particularly evident when considering the textual metrics: the bag-of-words source code representation which was found successful by Scandariato et al. [27] was instead poorly accurate in our case. This is true for both models exploiting this representation individually and those where textual features are combined with other metrics. This might be due to the fact that, when run on the set of modified files, the representation takes into account too many irrelevant tokens possibly creating noise, hindering the predictive capabilities of the bag-of-words to indicate the whether a commit contributes to a vulnerability. For this reason we also (1) employed the use of thresholds to discard both high- and low-frequency words, and (2) added the tokens extracted from the commit patch only, with the aim of reducing the noise and using more relevant features. Yet, these actions did not improve the overall quality of the textual metric set, highlighting the need for additional specific preprocessing activities aiming at further reducing noise. On a similar note, general-purpose code metrics alone often lead to poor results. For instance, the product metrics exploited in our study—and in vulnerability research in general—refer to the quantification of code quality aspects like cohesion, coupling, and complexity: while these have been successfully employed in other branches, e.g., code smell or defect prediction [108, 109], we observed that their contribution for just-in-time vulnerability detection is limited. Therefore, our results represent a call for new software metrics that can better characterize additional aspects of the source code, e.g., capturing security-related aspects [110, 111, 112], and evolutionary properties correlated to the presence of vulnerabilities.

Better together? On the combination of feature sets. As a follow-up discussion, it is worth analyzing the results achieved while combining multiple metrics. As recently reported by Theisen and Williams [31], vulnerability prediction models relying on a mixture of code, process, and textual metrics perform better than models based on individual features. When lowering the granularity of the prediction to commit-level, we found that this is not always the case, hence partially contrasting their results. As a matter of fact, there are some specific learning algorithms that appear to perform well under certain performance measures but fail when evaluated with different measures. For instance, despite showing very high AUC-ROC scores, the Random Forest models resulted to be one of the worst models in terms of F-measure when all the features were involved, apparently owing to the addition of textual metrics. At the same time, Theisen and Williams [31] show that the combination of textual and software metrics lead to a considerable drop in precision, hence affecting F-measure as well, in line with the results we observed in Figure 2. This suggests the need for automated mechanisms that can exploit contextual information to recommend which features would best fit the needs of the system where vulnerabilities must be diagnosed. A partial exception to this general finding was represented by process metrics: as shown in our study, these are the features that allow machine learners to significantly improve their detection capabilities. Our results seem to be in line with previous research showcasing the positive impact that change history information has on predictive modelling approaches [113, 114]. As such, it seems reasonable to argue that further research on the processes around the introduction of vulnerabilities should be performed to better characterize and improve their detection.

Ensemble learning for vulnerability prediction. In our investigation, we observed that the choice of the classifier has an impact on the resulting capabilities of just-in-time vulnerability detection models. While base learning algorithms typically have low performance, we noticed

that the use of ensemble methods improves the classification capabilities. On the one hand, this result does not come as a surprise, as ensemble learning has been introduced with the aim of overcoming the performance of base classifiers. On the other hand, however, it is also worth pointing out that previous investigations in the field of software engineering have revealed that the improvements given by ensemble methods might be limited when other aspects (e.g., availability of a balanced training set) come into play [115, 116]. Our findings specifically highlight that boosting methods might be promising for vulnerability detection and, indeed, the AdaBoost learner is the one obtaining the best performance. As observed in Section 4, its characteristics allow it to iteratively train a weak classifier on subsequent training data, assigning a weight to each instance of the training set, and leading to boost the learning capabilities. These results might drive practitioners in the selection of the technique to use when predicting vulnerabilities at commit-level, but also researchers to build upon these characteristics to engineer *ad-hoc* methodologies to further improve the boosting performance.

6. Threats to Validity

This section discusses the possible biases to our results and reports the employed mitigation strategies.

Threats to construct validity. A first threat in this category relates to the dataset exploited. We mined the *National Vulnerability Database* with the aim of collecting real, verified data on the vulnerabilities that affected software projects in the past. The nature of the information contained in NVD allowed us to be confident about the reliability of the dataset. Nonetheless, we cannot exclude imprecision: for instance, a patch reported in the database might have not removed a vulnerability as intended.

We relied on a technique based on SZZ to fetch the vulnerability-contributing commits that are likely to have caused the patch applied in the vulnerability-fixing commits mined from NVD. Previous studies have shown that this algorithm may frequently produce false positives [117]; to mitigate this risk we adopted some precautions. We exploited the implementation of SZZ provided by PyDriller [118], which follows the standard version of the algorithm [72] on which some adjustments have been included, i.e., discarding the candidate commits where only comments, cosmetic changes, or empty lines were blamed. This implementation achieved the highest recall with respect to the other variants [119], and so we opted for it to reduce the risk of missing relevant VCCs.

Over the initial population of 56,286 commits considered in our context, we sampled 8,991 commits due to computational constraints. We are aware that this sampling could have affected the performance of the machine learning models during the training and testing phases; however, our sampling criterion was carefully made random with the aim of mitigating the selection bias.

Another potential threat may be related to the selection of the independent variables used to build the experimented models. In this respect, we have carefully considered the related literature and the features previously used by researchers who targeted the problem of file-level vulnerability detection. Perhaps more importantly, our analyses targeted three different families of metrics, hence allowing us to experiment with features capturing different aspects of source code. Nevertheless, we cannot rule out that other metrics, not considered in the study, could provide additional contribution to the performance of just-in-time vulnerability detection methods. We plan to investigate this aspect further in the future.

Finally, when using the bag-of-words method we discarded the words appearing in over 80% of the documents (i.e., files or patches) or less than 5%. While this step could have removed some relevant features, and so possibly hindered the performance of the models, it is a recommended pre-processing step to remove noisy data and reduce the dimensionality of the dataset, which has been seen to have positive effects on the training process [77, 78]. The choice of these thresholds was directed by the need to have a reasonable number of features to train the models in an acceptable time without removing important tokens.

Threats to internal validity. In the context of our work, we selected and experimented eight machine learning models to better understand their strengths and weaknesses. Of course, the setting up of these approaches might have biased our results. However, we followed well-established guidelines [64, 38] through which we addressed possible issues due to multi-collinearity, missing hyper-parameter configuration, and data balancing issues. When focusing on these issues, we used methods and techniques that have been widely employed in the past (e.g., the `vif` function to deal with correlated variables) and that are recognized as effective.

Threats to external validity. Our study involved nine systems written in Java. On the one hand, we recognize that larger-scale studies would be desirable to further understand the capabilities of machine learning models for vulnerability detection. On the other hand, we are aware that different results might be obtained when addressing our research question on projects written in different programming languages or developed in different contexts (e.g., industrial systems). To enable replicability, we made all data and scripts available in our online appendix.[4] In any case, our future research agenda includes a large-scale replication of the study.

Threats to conclusion validity. To derive conclusive results on the performance of just-in-time vulnerability detectors, we first computed a number of evaluation metrics in an effort of capturing various angles of their capabilities. All of them uniformly indicated the poor performance of the experimented models, hence confirming our conclusions. In addition, we also applied statistical tests

to verify the significance of the differences observed: we run the Nemenyi rank test [105] to deal with the problem of multiple comparisons. This test is particularly useful in our context as it is suitable for non-normal distributions like the ones we experienced.

7. Conclusion

This paper proposed an empirical investigation into the capabilities of machine learning models for just-in-time vulnerability prediction. We took into account a set of eight machine learners and three families of features to provide a broad overview of how software vulnerabilities can be identified at commit-level.

Our key results indicated that the problem should be further investigated, as elaborated in Section 5. First, the currently available metrics seem to be not enough and, perhaps more importantly, their combination does not necessarily improve the detection capabilities. The research community should invest effort in defining empirical investigations into the features connected to the introduction of vulnerabilities at commit-level, other than the features that developers consider more relevant. For instance, we can envision the definition of longitudinal studies where developers are monitored for a given time period so that their activities might be closely analyzed in order to identify the key inducers of vulnerabilities. Similarly, we can envision studies aiming at elaborating catalogs of micro-antipatterns that developers frequently apply when contributing to vulnerabilities. An improved understanding of the features that more characterize the problem of software vulnerabilities would definitively improve the accuracy of just-in-time prediction models. On the basis of these empirical investigations, the definition of novel instruments able to compute those metrics and, perhaps more importantly, novel comprehensive datasets would be key to enable more and more research on the matter.

Second, our results indicate that the choice of the classifier impacts the performance: while most of the algorithms experimented achieve low F-measure scores, we observed that an ensemble method like AdaBoost seems to provide promising results that should be further analyzed and possibly improved by the research community. In other terms, our findings stimulate research targeting the engineering of software vulnerability prediction models. For instance, we could envision empirical studies and/or novel software engineering for artificial intelligence methods that could mix together the capabilities of individual classifiers or even dynamically adapt the classifier to use based on the peculiar characteristics of code commits and of the developers applying changes.

Last but not least, a collateral finding of our study concerns with the lack of public data and scripts that can be used to replicate/reproduce previous studies. This is pretty worrisome and allows us to recommend further research effort on the definition of standards and guidelines to make research reproducible, especially to enable researchers to compare the previous findings with new ones, hence leading to advance the state of the art in a safe and sustainable manner.

Our future research agenda includes a larger-scale replications of our study, other than the definition of novel techniques for (1) selecting features to use when identifying vulnerabilities at commit-level and (2) improving the training capabilities of ensemble approaches.

References

[1] M. Dowd, J. McDonald, J. Schuh, The art of software security assessment: Identifying and preventing software vulnerabilities, Pearson Education, 2006.

[2] G. McGraw, Software security, IEEE Security & Privacy 2 (2004) 80–83.

[3] A. Decan, T. Mens, E. Constantinou, On the impact of security vulnerabilities in the npm package dependency network, in: International Conference on Mining Software Repositories, pp. 181–191.

[4] H. Plate, S. E. Ponta, A. Sabetta, Impact assessment for vulnerabilities in open-source software libraries, in: International Conference on Software Maintenance and Evolution (ICSME), pp. 411–420.

[5] M. Finifter, D. Akhawe, D. Wagner, An empirical study of vulnerability rewards programs, in: 22nd {USENIX} Security Symposium ({USENIX} Security 13), pp. 273–288.

[6] S. Kim, H. Lee, Software systems at risk: An empirical study of cloned vulnerabilities in practice, Computers & Security 77 (2018) 720–736.

[7] D. Gonzalez, F. Alhenaki, M. Mirakhorli, Architectural security weaknesses in industrial control systems (ics) an empirical study based on disclosed software vulnerabilities, in: International Conference on Software Architecture (ICSA), pp. 31–40.

[8] I. Hydara, A. B. M. Sultan, H. Zulzalil, N. Admodisastro, Current state of research on cross-site scripting (xss)–a systematic literature review, Information and Software Technology 58 (2015) 170–186.

[9] D. R. McKinnel, T. Dargahi, A. Dehghantanha, K.-K. R. Choo, A systematic literature review and meta-analysis on artificial intelligence in penetration testing and vulnerability assessment, Computers & Electrical Engineering 75 (2019) 175–188.

[10] J. Svacina, J. Raffety, C. Woodahl, B. Stone, T. Cerny, M. Bures, D. Shin, K. Frajtak, P. Tisnovsky, On vulnerability and security log analysis: A systematic literature review on recent trends, in: International Conference on Research in Adaptive and Convergent Systems, pp. 175–180.

[11] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, J. Hu, Vulpecker: an automated vulnerability detection system based on code similarity analysis, in: Annual Conference on Computer Security Applications, pp. 201–213.

[12] M. Junjin, An approach for sql injection vulnerability detection, in: International Conference on Information Technology: New Generations, IEEE, pp. 1411–1414.

[13] A. I. Sotirov, Automatic vulnerability detection using static source code analysis, Ph.D. thesis, Citeseer, 2005.

[14] M.-T. Trinh, D.-H. Chu, J. Jaffar, S3: A symbolic string solver for vulnerability detection in web applications, in: Conference on Computer and Communications Security, pp. 1232–1243.

[15] H. Li, T. Kim, M. Bat-Erdene, H. Lee, Software vulnerability detection using backward trace analysis and symbolic execution, in: 2013 International Conference on Availability, Reliability and Security, IEEE, pp. 446–454.

[16] T. Wang, T. Wei, Z. Lin, W. Zou, Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution., in: NDSS, Citeseer.

[17] P. Saxena, P. Poosankam, S. McCamant, D. Song, Loop-extended symbolic execution on binary programs, in: International symposium on Software testing and analysis, pp. 225–236.
[18] B. Jiang, Y. Liu, W. Chan, Contractfuzzer: Fuzzing smart contracts for vulnerability detection, in: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp. 259–269.
[19] T. Wang, T. Wei, G. Gu, W. Zou, Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection, in: 2010 IEEE Symposium on Security and Privacy, IEEE, pp. 497–512.
[20] C. Vassallo, S. Panichella, F. Palomba, S. Proksc, H. Gall, A. Zaidman, How developers engage with static analysis tools in different contexts, Empirical Software Engineering (2019).
[21] N. Antunes, M. Vieira, Benchmarking vulnerability detection tools for web services, in: 2010 IEEE International Conference on Web Services, IEEE, pp. 203–210.
[22] L. N. Q. Do, J. Wright, K. Ali, Why do software developers use static analysis tools? a user-centered study of developer needs and motivations, IEEE Transactions on Software Engineering (2020).
[23] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in: International Conference on Software Engineering (ICSE), pp. 672–681.
[24] B. Chernis, R. Verma, Machine learning methods for software vulnerability detection, in: International Workshop on Security and Privacy Analytics, pp. 31–39.
[25] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, et al., Automated software vulnerability detection with machine learning, arXiv preprint arXiv:1803.04497 (2018).
[26] S. M. Ghaffarian, H. R. Shahriari, Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey, ACM Computing Surveys (CSUR) 50 (2017) 1–36.
[27] R. Scandariato, J. Walden, A. Hovsepyan, W. Joosen, Predicting vulnerable software components via text mining, IEEE Transactions on Software Engineering 40 (2014) 993–1006.
[28] Y. Shin, A. Meneely, L. Williams, J. A. Osborne, Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities, IEEE Transactions on Software Engineering 37 (2011) 772–787.
[29] T. Zimmermann, N. Nagappan, L. Williams, Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista, in: International Conference on Software Testing, Verification and Validation, pp. 421–428.
[30] C. Theisen, K. Herzig, P. Morrison, B. Murphy, L. Williams, Approximating attack surfaces with stack traces, in: International Conference on Software Engineering, volume 2, pp. 199–208.
[31] C. Theisen, L. Williams, Better together: Comparing vulnerability prediction models, Information and Software Technology 119 (2020) 106204.
[32] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, M. Harman, The importance of accounting for real-world labelling when predicting software vulnerabilities, in: European Software Engineering Conference and Symposium on the Foundations of Software Engineering, p. 695–705.
[33] G. Gousios, M. Pinzger, A. v. Deursen, An exploratory study of the pull-based software development model, in: International Conference on Software Engineering, pp. 345–355.
[34] J. Singer, T. Lethbridge, N. Vinson, N. Anquetil, An examination of software engineering work practices, in: CASCON First Decade High Impact Papers, 2010, pp. 174–188.
[35] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, A large-scale empirical study of just-in-time quality assurance, IEEE Transactions on Software Engineering 39 (2012) 757–773.
[36] J. Kang, D. Ryu, J. Baik, Predicting just-in-time software defects to reduce post-release quality costs in the maritime industry, Software: Practice and Experience 51 (2021) 748–771.
[37] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, A. Bacchelli, Information needs in contemporary code review, ACM on Human-Computer Interaction 2 (2018) 1–27.
[38] C. Tantithamthavorn, A. E. Hassan, An experience report on defect modelling in practice: Pitfalls and challenges, in: International Conference on Software Engineering: Software Engineering in Practice, pp. 286–295.
[39] R. M. O'brien, A caution regarding rules of thumb for variance inflation factors, Quality & quantity 41 (2007) 673–690.
[40] F. Pecorelli, D. Di Nucci, C. De Roover, A. De Lucia, A large empirical assessment of the role of data balancing in machine-learning-based code smell detection, Journal of Systems and Software (2020) 110693.
[41] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization., Journal of machine learning research 13 (2012).
[42] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: Conference on Computer and Communications Security, p. 529–540.
[43] K. Z. Sultana, V. Anu, T. Chong, Using software metrics for predicting vulnerable classes and methods in java projects: A machine learning approach, Journal of Software: Evolution and Process 33 (2020).
[44] P. Morrison, K. Herzig, B. Murphy, L. Williams, Challenges with applying vulnerability prediction models, in: Symposium and Bootcamp on the Science of Security.
[45] V. H. Nguyen, L. M. S. Tran, Predicting vulnerable software components with dependency graphs, in: International Workshop on Security Measurements and Metrics.
[46] I. Chowdhury, M. Zulkernine, Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities, Journal of Systems Architecture 57 (2011) 294–313.
[47] B. Smith, L. Williams, Using sql hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities, in: International Conference on Software Testing, Verification and Validation, p. 220–229.
[48] Y. Shin, L. Williams, Can traditional fault prediction models be used for vulnerability prediction?, Empirical Software Engineering 18 (2011) 25–59.
[49] J. Walden, J. Stuckman, R. Scandariato, Predicting vulnerable components: Software metrics vs text mining, in: International Symposium on Software Reliability Engineering, pp. 23–33.
[50] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, S. Li, Combining software metrics and text features for vulnerable file prediction, in: International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 40–49.
[51] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, Y. Acar, Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits, in: ACM SIGSAC Conference on Computer and Communications Security, p. 426–437.
[52] L. Yang, X. Li, Y. Yu, Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes, in: IEEE Global Communications Conference, pp. 1–7.
[53] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on software engineering 20 (1994) 476–493.
[54] T. J. McCabe, A complexity measure, IEEE Transactions on software Engineering (1976) 308–320.
[55] Y. Shin, L. Williams, An empirical model to predict security vulnerabilities using code complexity metrics, in: International Symposium on Empirical Software Engineering and Measurement, p. 315–317.
[56] Y. Zhang, R. Jin, Z.-H. Zhou, Understanding bag-of-words model: a statistical framework, International Journal of Machine Learning and Cybernetics 1 (2010) 43–52.
[57] Z. S. Harris, Distributional structure, Word 10 (1954) 146–162.

[58] J. Yang, K. Hotta, Y. Higo, H. Igaki, S. Kusumoto, Classification model for code clones based on machine learning, Empirical Softw. Engg. 20 (2015) 1095–1125.
[59] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: A benchmark and an extensive comparison, Empirical Softw. Engg. 17 (2012) 531–577.
[60] N. Bhatt, A. Anand, V. Yadavalli, Exploitability prediction of software vulnerabilities, Quality and Reliability Engineering 37 (2020).
[61] T. Riom, A. Sawadogo, K. Allix, T. F. Bissyandé, N. Moha, J. Klein, Revisiting the vccfinder approach for the identification of vulnerability-contributing commits, Empirical Software Engineering 26 (2021) 46.
[62] Cwe-119: Improper restriction of operations within the bounds of a memory buffer, `https://cwe.mitre.org/data/definitions/119.html`, 2006. Online; accessed 02 December 2021.
[63] V. R. Basili, G. Caldiera, H. D. Rombach, The goal question metric approach, Encyclopedia of Software Engineering (1994).
[64] Q. Song, Z. Jia, M. Shepperd, S. Ying, J. Liu, A general software defect-proneness prediction framework, IEEE transactions on software engineering 37 (2010) 356–370.
[65] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, A. Wesslen, Experimentation in Software Engineering: An Introduction, 2000.
[66] P. R. et al., Empirical standards for software engineering research, 2021.
[67] NIST, U.s. nist computer security division, `https://www.nist.gov`, 2021. Online; accessed 06 July 2021.
[68] MITRE, Common vulnerabilities and exposures, `https://cve.mitre.org/`, 2021. Online; accessed 06 July 2021.
[69] O. Alhazmi, Y. Malaiya, I. Ray, Measuring, analyzing and predicting security vulnerabilities in software systems, Computers & Security 26 (2007) 219–228.
[70] S. Huang, H. Tang, M. Zhang, J. Tian, Text clustering on national vulnerability database, in: International Conference on Computer Engineering and Applications, volume 2, pp. 295–299.
[71] S. Zhang, D. Caragea, X. Ou, An empirical study on using the national vulnerability database to predict software vulnerabilities, in: International Conference on Database and Expert Systems Applications, pp. 217–231.
[72] J. Sliwerski, Z. T., A. Zeller, When do changes induce fixes?, in: International Workshop on Mining Software Repositories, MSR '05, pp. 1–5.
[73] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, B. Spates, When a patch goes bad: Exploring the properties of vulnerability-contributing commits, in: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 65–74.
[74] Git - git-merge documentation, `https://git-scm.com/docs/git-merge`, 2021. Online; accessed 19 November 2021.
[75] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, S. Panichella, Applying a smoothing filter to improve ir-based traceability recovery processes: An empirical investigation, Information and Software Technology 55 (2013) 741–754.
[76] C. Silva, B. Ribeiro, The importance of stop word removal on recall values in text categorization, in: Proceedings of the International Joint Conference on Neural Networks, 2003., volume 3, pp. 1661–1666 vol.3.
[77] C. A. Martins, Reducing the dimensionality of bag-of-words text representation used by learning algorithms.
[78] T. Joachims, Text categorization with support vector machines: Learning with many relevant features, in: C. Nédellec, C. Rouveirol (Eds.), Machine Learning: ECML-98, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 137–142.
[79] A. Meneely, O. Williams, Interactive churn metrics: Socio-technical variants of code churn, SIGSOFT Softw. Eng. Notes 37 (2012) 1–6.
[80] N. Nagappan, T. Ball, Use of relative code churn measures to predict system defect density, in: International Conference on Software Engineering, 2005. ICSE 2005., pp. 284–292.
[81] T. Graves, A. Karr, J. Marron, H. Siy, Predicting fault incidence using software change history, IEEE Transactions on Software Engineering 26 (2000) 653–661.
[82] M. Piancó, B. Fonseca, N. Antunes, Code change history and software vulnerabilities, in: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), pp. 6–9.
[83] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad (and whether the smells go away), IEEE Transactions on Software Engineering 43 (2017) 1063–1088.
[84] N. Nagappan, B. Murphy, V. Basili, The influence of organizational structure on software quality: An empirical case study, in: International Conference on Software Engineering, p. 521–530.
[85] D. L. Parnas, Software aging, in: International Conference on Software Engineering, ICSE '94, IEEE Computer Society Press, Washington, DC, USA, 1994, p. 279–287.
[86] V. Bandara, T. Rathnayake, N. Weerasekara, C. Elvitigala, K. Thilakarathna, P. Wijesekera, C. Keppitiyagama, Fix that fix commit: A real-world remediation analysis of javascript projects, in: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 198–202.
[87] K. Herzig, S. Just, A. Zeller, The impact of tangled code changes on defect prediction models, Empirical Software Engineering 21 (2016) 303–336.
[88] A. E. Hassan, Predicting faults using the complexity of code changes, in: 2009 IEEE 31st International Conference on Software Engineering, pp. 78–88.
[89] M. Tufano, G. Bavota, D. Poshyvanyk, M. D. Penta, R. Oliveto, A. D. Lucia, An empirical study on developer-related factors characterizing fix-inducing commits, Journal of Software: Evolution and Process 29 (2017).
[90] A. G. Koru, D. Zhang, K. El Emam, H. Liu, An investigation into the functional form of the size-defect relationship for software modules, IEEE Transactions on Software Engineering 35 (2009) 293–304.
[91] B. Henderson-Sellers, L. Constantine, I. Graham, Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design), Object Oriented Syst. 3 (1996) 143–158.
[92] C. Cortes, V. Vapnik, Support-vector networks, Machine learning 20 (1995) 273–297.
[93] Z. Zhang, Introduction to machine learning: k-nearest neighbors, Annals of translational medicine 4 (2016).
[94] L. Breiman, J. Friedman, C. J. Stone, R. Olshen, Classification and regression trees Regression trees, Chapman and Hall, 1984.
[95] L. Breiman, Random forests, Machine learning 45 (2001) 5–32.
[96] P. Geurts, D. Ernst, L. Wehenkel, Extremely randomized trees, Machine Learning 63 (2006) 3–42.
[97] Y. Freund, R. E. Schapire, A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting, Journal of Computer and System Sciences 55 (1997) 119–139.
[98] R. E. Schapire, The Strength of Weak Learnability, Machine Learning 5 (1990) 197–227.
[99] J. H. Friedman, Greedy function approximation: A gradient boosting machine., The Annals of Statistics 29 (2001) 1189 – 1232.
[100] T. Chen, C. Guestrin, XGBoost: A Scalable Tree Boosting System, in: 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16, pp. 785–794.
[101] B. Coppin, Artificial intelligence illuminated, Jones & Bartlett Learning, 2004.
[102] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, Smote: Synthetic minority over-sampling technique, J. Artif. Int. Res. 16 (2002) 321–357.
[103] D. M. W. Powers, Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation, Journal of Machine Learning Technologies 2 (2011) 37–63.

[104] B. Matthews, Comparison of the predicted and observed secondary structure of t4 phage lysozyme, Biochimica et Biophysica Acta (BBA) - Protein Structure 405 (1975) 442–451.

[105] P. Nemenyi, Distribution-free multiple comparisons, in: Biometrics, volume 18, International Biometric Soc 1441 I ST, NW, SUITE 700, WASHINGTON, DC 20005-2210, p. 263.

[106] Flawfinder, `https://dwheeler.com/flawfinder`, 2021. Online; accessed 24 November 2021.

[107] M. Shepperd, D. Bowes, T. Hall, Researcher bias: The use of machine learning in software defect prediction, IEEE Transactions on Software Engineering 40 (2014) 603–616.

[108] M. I. Azeem, F. Palomba, L. Shi, Q. Wang, Machine learning techniques for code smell detection: A systematic literature review and meta-analysis, Information and Software Technology 108 (2019) 115–138.

[109] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering 38 (2011) 1276–1304.

[110] B. Alshammari, C. Fidge, D. Corney, A hierarchical security assessment model for object-oriented programs, in: 2011 11th International Conference on Quality Software, pp. 218–227.

[111] I. Chowdhury, B. Chan, M. Zulkernine, Security metrics for source code structures, in: Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems, SESS '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 57–64.

[112] A. Younis, Y. Malaiya, C. Anderson, I. Ray, To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit, in: Conference on Data and Application Security and Privacy, p. 97–104.

[113] L. Pascarella, F. Palomba, A. Bacchelli, Fine-grained just-in-time defect prediction, Journal of Systems and Software 150 (2019) 22–36.

[114] F. Rahman, P. Devanbu, How, and why, process metrics are better, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, pp. 432–441.

[115] I. H. Laradji, M. Alshayeb, L. Ghouti, Software defect prediction using ensemble learning on selected features, Information and Software Technology 58 (2015) 388–402.

[116] F. Pecorelli, D. Di Nucci, Adaptive selection of classifiers for bug prediction: A large-scale empirical analysis of its performances and a benchmark study, Science of Computer Programming 205 (2021) 102611.

[117] G. Rodríguez-Pérez, G. Robles, J. M. González-Barahona, Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm, Information and Software Technology 99 (2018) 164–176.

[118] D. Spadini, M. Aniche, A. Bacchelli, PyDriller: Python framework for mining software repositories, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018, ACM Press, New York, New York, USA, 2018, pp. 908–911.

[119] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, R. Oliveto, Evaluating szz implementations through a developer-informed oracle, 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (2021) 436–447.