

An Empirical Study on the Performance of Vulnerability Prediction Models Evaluated Applying Real-world Labelling

Giulia Sellitto¹, Alexandra Sheykina¹, Fabio Palomba¹ and Andrea De Lucia¹

¹Software Engineering (SeSa) Lab, University of Salerno, Via Giovanni Paolo II, 132 - 84084 Fisciano (Salerno), Italy

Abstract

Software vulnerabilities are infamous threats to the security of computing systems, and it is vital to detect and correct them before releasing any piece of software to the public. Many approaches for the detection of vulnerabilities have been proposed in the literature; in particular, those leveraging machine learning techniques, *i.e.*, vulnerability prediction models, seem quite promising. However, recent work has warned that most models have only been evaluated in *in-vitro* settings, under certain assumptions that do not resemble the real scenarios in which such approaches are supposed to be employed. This observation ignites the risk that the encouraging results obtained in previous literature may be not as well convenient in practice. Recognizing the dangerousness of biased and unrealistic evaluations, we aim to dive deep into the problem, by investigating whether and to what extent vulnerability prediction models' performance changes when measured in realistic settings. To do this, we perform an empirical study evaluating the performance of a vulnerability prediction model, configured with three data balancing techniques, executed at three different degrees of realism, leveraging two datasets. Our findings highlight that the outcome of any measurement strictly depends on the experiment setting, calling researchers to take into account the actuality and applicability in practice of the approaches they propose and evaluate.

Keywords

Vulnerability Prediction, Realistic Evaluation, Empirical Study

1. Introduction

Software vulnerabilities are flaws or oversights in a piece of software that allow attackers to do something malicious, *e.g.*, expose or alter sensitive information, disrupt or destroy a system, or take control of a computer program [1]. Many infamous cases of vulnerabilities being exploited are reported every year; the most notorious remain those leaking private users' data and causing monetary losses in the millions. It is vital for software developers to release secure systems; therefore, any vulnerability affecting the code must be found and corrected before production. The activity of discovering security flaws in software is known as *vulnerability discovery*, and a plethora of approaches have been proposed in the literature, leveraging static, dynamic, or

Joint Conference of the 32nd International Workshop on Software Measurement (IWSM) and the 17th International Conference on Software Process and Product Measurement (MENSURA), September 14–15, 2023, Rome, Italy


✉ gisellitto@unisa.it (G. Sellitto); asheykina@unisa.it (A. Sheykina); fpalomba@unisa.it (F. Palomba); adelucia@unisa.it (A. De Lucia)

🌐 <https://giuliasellitto7.github.io/> (G. Sellitto); <https://fpalomba.github.io/> (F. Palomba);

<https://docenti.unisa.it/003241/en/home> (A. De Lucia)

🆔 0000-0002-5491-0873 (G. Sellitto); 0000-0001-9337-5116 (F. Palomba); 0000-0002-4238-1425 (A. De Lucia)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

hybrid analysis [2, 3, 4]. More recently, machine learning and deep learning have been applied to the task of finding vulnerabilities; in such cases, the activity is called *vulnerability prediction*. Leveraging machine learning algorithms, any piece of software can be labelled as *vulnerable* or *neutral*, whether it is affected by a known vulnerability or not (yet). Many vulnerability prediction models (VPMs) have been defined in the literature [5, 6], each operating at different granularity, *i.e.*, revision- (or commit-) level, file- (or class-) level, or function- (or method-) level, and leveraging different pieces of information, *i.e.*, structural properties of the code, textual features, or amount of modifications performed over time. The performance of such VPMs seems promising, hinting at their expected large utility in software development life cycles.

However, Jimenez *et al.* [7] have recently alerted the research community working on VPMs, warning about the importance of taking into account the degree of realism under which such models are trained and evaluated. They warned that evaluations are often performed in *in-vitro* settings which are not properly realistic, and lead to biased results, as they do not align with the real-world scenarios in which VPMs are supposed to be employed.

The ideal usage of such models consists in leveraging vulnerability data collected through the history of the project evolution to predict the flaws threatening the current version being developed. Nevertheless, most VPMs have been evaluated in the literature via the well-known cross-validation strategy, which allows a fair assessment of their performance, but splits data into folds without considering the time relationships among them. Indeed, in every but one round of cross-validation, data from the future is used to train the model, which is then tested against data from the past; this does not resemble the actual usage scenario of VPMs, and makes the evaluation unhelpful in practice.

Perhaps more concerning, evaluations of VPMs are performed under what Jimenez *et al.* called the *perfect labelling assumption* [7], *i.e.*, supposing that all vulnerabilities known from time t onwards are available at all times, even before t . This practically translates into the fact that data used for training and testing are labelled according to an oracle that is accessible to researchers at evaluation time. However, in real-world scenarios, new vulnerabilities are discovered as software systems evolve over time; therefore, it is not guaranteed that at any time every vulnerability has already been discovered and localized in the affected code.

Recognizing the dangerousness of biased and unrealistic evaluations, Jimenez *et al.* [7] exercised VPMs in real-world scenarios, and found that the performance of such models, although seeming promising at first glance, drop significantly when evaluated in realistic settings. This issue is a severe threat potentially invalidating all the effort spent by the research community on the proposal and evaluation of VPMs, as they risk not being applicable in practice, and therefore result useless.

Our goal is to dive deep into the problem, by investigating whether and to what extent VPMs' performance changes when evaluated in realistic settings. To do this, we perform an empirical study exercising a well-known vulnerability prediction model at three different degrees of realism, to analyse whether it would be suitable in real-world scenarios. First, we operate at zero realism, adopting the perfect labelling assumption, *i.e.*, we run the experiments in a flawless *in-vitro* setting. Afterwards, we take a small step toward proper realism, by applying a *release-based* evaluation strategy, considering the time relationship between past and future data, but still under the perfect labelling assumption, *i.e.*, leveraging the knowledge of an oracle that we, as researchers, can count on. Finally, we exercise the VPM performing a release-based

evaluation applying *real-world labelling* on training and testing data, to wear practitioners' shoes and leverage the only information that is available at training time to label data samples. We leverage the model and dataset by Walden *et al.* [8] and follow the ideas and recommendations of Jimenez *et al.* [7] to evaluate the performance of the well-known VPM based on twelve source code metrics, configured with three data balancing techniques, using two software projects written in PHP as datasets.

We find out that the performance of VPMs drop considerably when they are evaluated outside the *in-vitro* context, highlighting the fact that the outcome of a measurement strictly depends on the experiment setting. Therefore, we encourage researchers working on VPMs to take into account the true applicability in practice of the proposed approaches, to make research efforts more meaningful in the implementation context, and to enhance the cooperation between academia and industry.

Structure of the paper. This paper is organized as follows. In Section 2, we introduce the background concepts involved in our study, along with relevant literature on the matter. In Section 3, we report the details of our study design following the Goal-Question-Metric (GQM) template [9], and we discuss the analysis of the results in Section 4. We further elaborate on our findings in Section 5, extracting meaningful take-away messages for the communities of researchers and practitioners. We recognize the threats to the validity of our work in Section 6, reporting the strategies we followed to mitigate them. Finally, in Section 7, we conclude the paper drawing a summary of our investigation, and we make all the data, scripts, and results of our study available in Appendix A.

2. Background and Related Work

Much work has been done by researchers to address the problem of detecting vulnerabilities in software, adopting different approaches and techniques. Traditional methods such as static analysis, dynamic analysis, and hybrid analysis are used, as well as machine learning techniques for the *vulnerability prediction* task [5]. One of the first studies that investigated a machine learning approach for vulnerability prediction was by Neuhaus *et al.* [10]; they designed a tool, VULTURE, which predicts vulnerabilities in C/C++ functions.

Most vulnerability prediction models proposed over the years operate at *file-level granularity*, leveraging different pieces of information to feed the model [5]. Meneely and Williams [11, 12] considered developer activity metrics as predictors of the presence of security vulnerabilities in the source code of the LINUX KERNEL, the PHP programming language and the WIRESHARK network analyzer. They investigated the Linus' Law defined by Raymond [13] about the number of developers involved in the software project, and found that developer activity metrics can be used as indicators of the vulnerability of the source code file. They extended such findings by working with Shin [14] to investigate the usage of execution complexity metrics along with code churn and developer metrics as indicators of software vulnerabilities. They conducted their empirical studies on two open-source projects, *i.e.*, RED HAT LINUX and MOZILLA FIREFOX web browser, and found out that the leveraged metrics exhibited significant discriminative power over the prediction of vulnerable files.

Zimmerman *et al.* [15] carried out an empirical study to further extend the feature set to be

employed for vulnerability prediction, by considering program complexity, code churn, test coverage, dependency measures, and organizational structure of the company in the context of WINDOWS VISTA. They observed that dependency metrics led to significantly high performance in terms of recall values, complementing the weaknesses of other features computed on the source code, such as complexity measures.

Walden *et al.* [8] proposed two models based on source code metrics and textual features, respectively, and evaluated them by performing an empirical study on a vulnerability dataset collected from three open-source PHP web applications *i.e.*, PHPMYADMIN, MOODLE, and DRUPAL, containing 223 vulnerabilities. The latter VPM uses the *Bag of Words* approach, which the authors applied in previous work along with Hovsepyan [16]. Walden *et al.* compared the performance obtained from the two models evaluated with 3-fold cross-validation, and obtained encouraging results. Kaya *et al.* [17] operated on the same dataset to evaluate the impact of choosing different settings when building a model. They compared the performance obtained considering seven classifiers and four data balancing techniques to deal with the imbalance of the dataset. They employed Walden *et al.* models, using software metrics, text tokens, and a combination of the two as predictors, and observed that data balancing methods are effective for highly unbalanced datasets, and the Random Forest classifier is most performing on small datasets.

Song *et al.* [18] investigated the effect of biased learning and its interactions with data bias, classifier type, and input metrics. They concluded that unbalanced learning should only be considered for moderate or highly unbalanced data sets, and the indiscriminate application of imbalanced learning can be detrimental. Wu *et al.* [19] analyzed the impact of the class imbalance problem of security bug report prediction and confirmed its negative impact on prediction performance; they performed a comparative study on six balancing methods combined with five popular classification algorithms. Zhang *et al.* [20] experimented with the utilization of the two predictors, *i.e.*, source code metrics and textual features, jointly, proposing an original approach called VULPREDICTOR. An additional multi-level solution was proposed by Catal *et al.* [21], who deployed a vulnerability prediction web service on the MICROSOFT AZURE cloud computing platform. The service takes software metrics as predictors and, after performing steps of data cleaning and preparation, it feeds data to a stratified neural network for vulnerability prediction. Recent successes in natural language processing (NLP) techniques have encouraged research into learning representation for source code, which relies on similar NLP methods for identifying vulnerable code [22]. Since vulnerabilities are a specific case of software defects, *i.e.*, defects threatening the security of programs, defect prediction approaches proposed in the literature over the years have been also applied to the task of predicting software vulnerabilities [23], and encouraging results have been achieved [24, 25, 26, 27, 28, 29]. Additional solutions to the vulnerability prediction problem consisted of original approaches, *e.g.*, TROVON, proposed by Garg *et al.* [30]. They developed a prediction method using the machine translation encoder/decoder framework that automatically learns the code latent features linked to the vulnerabilities. They performed release-based experiments on the Linux Kernel, Wireshark, and OpenSSL datasets with realistic training data settings.

Scandariato *et al.* [31] investigated whether and to what extent mobile applications developed for the Android platform are affected by vulnerabilities, and how it would be possible to predict which classes are compromised, by analyzing the application on the Android store and developing a vulnerability prediction model, which exhibited high accuracy (over 0.8).

They focused their work on 20 Android applications and employed the *Bag of Words* method based on text tokens. They also investigated *release-based* validation approaches in subsequent research [32], using past data to train the model, and using future data to test it against. Also Jimenez *et al.* [33], in a previous study on the LINUX KERNEL dataset, applied a *release-based* validation method, and found out that the performance drop when taking into account the time relationships existing among data.

More recently, Jimenez *et al.* [7] highlighted a problem arising from the application of cross-validation to evaluate vulnerability prediction models. They argued that researchers work under the so-called *perfect labelling assumption*, *i.e.*, they assume, unconsciously or indirectly, that vulnerability data is always available. This is due to the fact that once a vulnerability is known to affect a file, that file in the dataset is labelled as vulnerable from the time at which the vulnerability was introduced in the source code onwards. In a real context, this is not feasible: vulnerabilities are discovered or reported only after a certain period of time, that is subsequent to the moment they have been introduced. Hence, when evaluating a prediction model, one should use *real-world labelling* and consider training the model at a certain time t , using only the data available at t , *i.e.*, vulnerabilities that have already been discovered before t . Jimenez *et al.* compared the performance output by vulnerability prediction models under the perfect labelling assumption with the performance obtained when considering *real-world labelling* and a *release-based* validation approach. They discovered that, when evaluated in a scenario that is more similar to the real operating context, vulnerability prediction models do not perform as well as one would wish. They showed significantly lower predictive effectiveness (mean Matthews Correlation Coefficient values of 0.08, 0.22 and 0.10 were achieved for LINUX, OPENSLL and WIRESARK, respectively) when models are trained only on vulnerability labels that could realistically be available to the practitioners at the time of model building.

Following the path traced by Jimenez *et al.* [7], Sellitto *et al.* [34] recently analyzed the impact of using a release-based validation approach on vulnerability prediction models. They confirmed that taking into account the time relationship existing among data has a considerable impact on the performance of VPMs leading to generally lower performance, highlighting that further research would be needed to make vulnerability prediction models more effective.

Since vulnerabilities are a subset of software defects, the considerations risen by Jimenez *et al.* [7] have been embraced also in the context of defect prediction. Bangash *et al.* [35] evaluated five cross-project defect prediction approaches, and showed that data belonging to different time periods generates varying results. They applied a *time-aware* evaluation approach, in which models are trained only on the past data, and evaluations are executed only on the subsequent data. In previous research, Huang *et al.* [36] and Yang *et al.* [25] used an approach called *time-wise cross-validation* for defect prediction, considering the release order, but ignoring different time periods, leveraging future data that would not be available at the time of training the model. Tan *et al.* [37] argued that ignoring time leads to highly unrealistic performance estimates in defect prediction scenarios, providing support to the findings observed in the vulnerability prediction contexts.

Research Gap and Our Work. Existing literature has shown that vulnerability prediction models exhibit encouraging performance when evaluated under the *perfect labelling assumption*. However, little is known about their true applicability in real-world software development

scenarios [7]. We take a step to fill this gap by investigating one of the most well-known VPMs in the literature, *i.e.*, the model based on source code metrics proposed by Walden *et al.* [8]. We follow the recommendations of Jimenez *et al.* [7] to evaluate how the performance of the model changes when evaluated in more realistic settings. We extend our previous work on the matter [34] by experimenting with a release-based validation applying realistic labelling. We contribute to the state-of-the-art by broadening the set of VPMs that have been exercised taking into account the realistic availability of vulnerability data, augmenting the body of knowledge on their true applicability in practice.

3. Study Design

Our *goal* is to investigate how vulnerability prediction models' performance change when evaluated in realistic settings, with the *purpose* of understanding the deviation of their expected performance from their actual usefulness in real scenarios. The *perspective* is of both practitioners and researchers; the former are interested in realising how the experimental settings influence the observed results when evaluating an approach for vulnerability prediction, and the latter are concerned about the true applicability of such approaches in their production. The *context* of our study is given by the file-level vulnerability prediction models and dataset by Walden *et al.* [8]. The VPM that we use is based on source code metrics; the dataset we leverage contains a total of 126 vulnerabilities affecting multiple versions of two open-source web applications written in PHP, *i.e.*, PHPMYADMIN, and MOODLE.

First, we are interested in assessing the performance of VPMs in the *perfect* scenario, *i.e.*, at *zero* degree of realism. In this way, we lay down the baseline needed for the comparison, as the main goal of our empirical study is to look into the difference in the observed performance in different evaluation settings. Thus, we ask:

Q RQ₁. *What is the performance of vulnerability prediction models evaluated via cross-validation under the perfect-labelling assumption?*

To take a step toward realism, a *release-based* validation approach can be considered, in which data from previous releases of software is used to train the model, and data from the next release is used to test the model against to. Such a validation method takes into account the low degree of realism provided by cross-validation, and overcomes it by considering the relationship between past and present data. This approach is more similar to what developers would do in real scenarios, *i.e.*, they would leverage information coming from the history of the project to understand the possible threats to the current version being developed. We want to assess whether and how such validation strategy leads to significantly different performance than the perfect scenario; therefore, we ask:

Q RQ₂. *What is the performance of vulnerability prediction models evaluated via release-based validation under the perfect-labelling assumption?*

As suggested by Jimenez *et al.* [7], a fully-realistic evaluation approach must take into account the availability of vulnerability data release by release, tailoring the labelling of the training and

test set accordingly. As vulnerabilities are discovered over time, instances must be labelled as *vulnerable* or *neutral* based on the time the evaluation is set; *e.g.*, an evaluation round considering the first, second, and third release of a software as training set and the fourth release as the test set, must not label as *vulnerable* those instances whose vulnerabilities have been discovered after the fourth release. Such a validation strategy resembles a realistic scenario, in which no information on the vulnerabilities that will be discovered in the future would be available yet. We want to assess the performance of VPMs in the fully realistic setting, thus, we ask:

Q RQ₃. *What is the performance of vulnerability prediction models evaluated via release-based validation with real-world labelling?*

By answering our three research questions, we aim at understanding whether the performance of VPMs that have been demonstrated in the literature are confined to *in-vitro* settings, or they can be effectively leveraged in real software development scenarios. In adherence to open science principles, we make all the data, scripts, and results of our study available in Appendix A.

3.1. Context

The context of our empirical study is given by the PHP vulnerability dataset proposed by Walden *et al.* [38], which contains vulnerability data mined from three popular open-source PHP applications, *i.e.*, PHPMYADMIN, MOODLE and DRUPAL. The dataset was published along with two file-level vulnerability prediction models [8], one based on source code metrics, and one based on text tokens. For our research purposes, we leverage a selection of such dataset and models, namely, we do not consider the model based on text tokens and the DRUPAL project in our experiments, as we are not able to access time-related information that is necessary to answer to our research questions.

We perform our work on the model based on source code metrics, using the dataset consisting of 95 releases of PHPMYADMIN and 71 releases of MOODLE, reported to be affected by 75 and 51 vulnerabilities, respectively, described in detail in Table 1. The dataset provides a tracking matrix that keeps a record of which files were affected by each vulnerability, at the time of each considered release. Some vulnerabilities migrated among versions, as time passed from their introduction in the code to their discovery and fix.

The model based on source code metrics [8] is built using twelve file-level features extracted from each file belonging to each considered release of the two software projects; the set of metrics is reported in Table 2. Each file of the software system under analysis is labelled as *vulnerable* if it contains at least one vulnerability, or as *neutral* if it is known to contain no vulnerabilities at the moment. It is worth pointing out that files are never labelled as *non-vulnerable*, since we cannot be sure that they do not contain any vulnerabilities at all. Indeed, they could contain vulnerabilities that have not been discovered yet.

3.2. Experimental Settings

To answer our research questions, we perform experiments in three different settings.

First, to answer **RQ₁**, we operate at *zero* degree of realism, arranging a purely *in-vitro* experimental setting. We evaluate the vulnerability prediction model via cross-validation under

Vulnerability Type	Description	Number of Samples	
		PHPMyAdmin	Moodle
Code Injection	Allow attackers to modify server-side variables or HTTP headers, or execute code on the server.	10	7
CSRF	Induce the authorized user to perform unintended actions the attacker wants.	1	3
XSS	Allow malicious Javascript code to be executed in the browser of the user.	45	9
Path Disclosure	Allow malicious exploiters to obtain the installation path of the application; this information can be useful to perform a subsequent attack.	12	2
Authorization Issues	General violations of the CIA triad, <i>i.e.</i> , confidentiality, integrity, or availability.	6	28
Other	No better-specified vulnerability.	1	2
Total		75	51
		126	

Table 1
Distribution of vulnerability types in the leveraged dataset [8].

the *perfect-labelling* assumption. Following the original work by Walden *et al.* [8], we perform three-fold cross-validation, which consists in dividing the dataset into three equally-large segments, *i.e.*, folds, and running three rounds of evaluation. In each round, data belonging to two folds are used to train the model, and data from the remaining fold are used to test the model; this approach ensures that the VPM is tested against each sample in the dataset once, and the overall performance is computed as the average results of the three rounds. The *perfect-labelling* assumption consists in using the complete dataset provided with the labels assigned to the samples at the time it was built. Time relationships existing among data are not considered in this first experimental setting, in which we operate similarly to most previous work in the field of vulnerability prediction [7].

To answer **RQ₂**, we take a step toward realism, considering the time relationships existing among data. Rather than cross-validation, we apply a *release-based* validation strategy, following prior work by Shin *et al.* [14, 23]. In such a strategy, the validation is performed in rounds and is based on software releases. In each round, data belonging to a single release R_i is used as the test set, and data from the three immediately prior releases R_{i-3} , R_{i-2} , and R_{i-1} is leveraged as the training set. Thus, we start by using R_1 , R_2 , and R_3 for training and R_4 for testing, and we proceed as depicted in Figure 1, resulting in the execution of several experiments per dataset, depending on the number of releases, *i.e.*, 95 for PHPMYADMIN, and 71 for MOODLE. In this experimental setting, we still label data leveraging the available knowledge at the time the dataset was collected. In this way, we impersonate researchers who are concerned about the time relationships among data, still operating in an *in-vitro* setting.

Finally, to answer **RQ₃**, we wear the shoes of practitioners to operate in a more realistic

Metric	Description
Lines of Code	The number of lines in a source file where PHP tokens occur. Lines not containing PHP tokens, such as blank lines and comments, are excluded from the count.
Lines of Code (non-HTML)	Lines of code, except HTML content embedded in PHP files, <i>i.e.</i> , content outside of PHP start and end tags, is not considered.
Number of Functions	The number of function and method definitions in a file.
Cyclomatic Complexity	The size of a control flow graph after linear chains of nodes are collapsed into one.
Maximum Nesting Complexity	The maximum depth to which loops and control structures in the file are nested.
Halstead's Volume	Estimated as the file's vocabulary size multiplied by the logarithm of the file length. The vocabulary size is given by the sum of the number of unique operators and unique operands. Operators are method names and PHP language operators, operands are parameter and variable names. The file length is given by the sum of the total number of operators and operands.
Total External Calls	The number of times a statement in the file invokes a function or method defined in a different file.
Fan-in	The number of other files that contain statements that invoke a function or method defined in the file being measured.
Fan-out	The number of other files that contain functions or methods invoked by statements of the file being measured.
Internal Functions or Methods Called	The number of functions or methods defined in the file that are called at least once by a statement of the same file.
External Functions or Methods Called	The number of functions or methods defined in other files which are called at least once by a statement in the file being measured.
External Calls to Functions or Methods	The number of files that contains statements calling a particular function or method defined in the file being measured, summed across all functions and methods of the file being measured.

Table 2

Source code metrics used as features of the vulnerability prediction model [8].

scenario. Developers willing to employ VPMs in software production can only rely on partial knowledge, consisting uniquely of the information available at the time the model is trained. In particular, the labelling of the samples both in the training and testing set is subject to the time of discovery of vulnerabilities; files can only be labelled as *vulnerable* if a vulnerability has already been discovered at the labelling time, that is, model training time. This rationale leads to the definition of the *real-world* labelling approach [7], that we employ in the third experimental setting, together with the *release-based* evaluation strategy. In each round of the *release-based* validation, we build an *ad-hoc* dataset, making sure that only data available at training time is leveraged. In particular, for each release R_i picked as a test set, we get the release date D_{R_i} ; since the software was ready to be published at that time, we can assume that a VPM could be run on that date on the version about to be released; therefore, we consider D_{R_i} as the training time. We then label the data in both the training and test set according to the

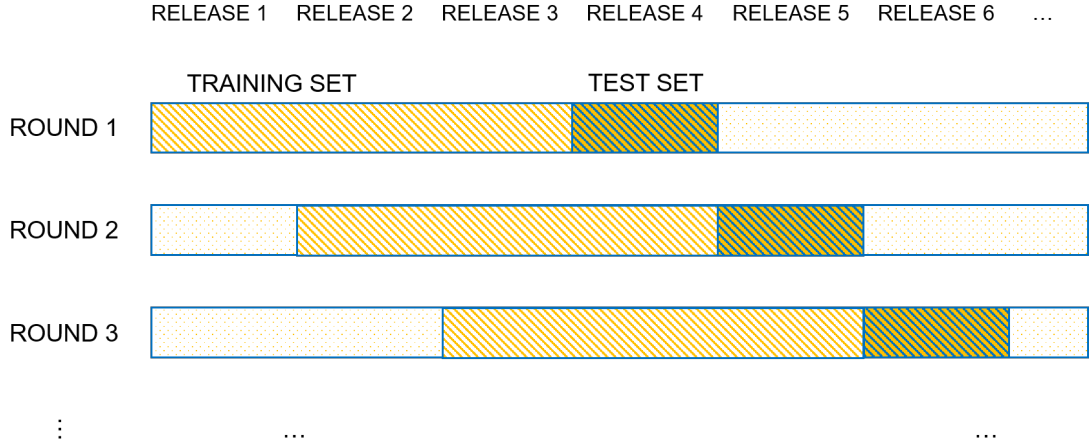


Figure 1: The *release-based* validation approach [23, 14].

information available at time D_{R_i} . Namely, we label samples as *vulnerable* if and only if they are affected by a vulnerability whose publication time is prior to D_{R_i} ; to get this information, we leverage the CVE-SEARCH service. This third experimental setting further resembles the real-world scenario VPMs would be employed, and provides us with a less biased evaluation of their actual performance.

In all three experimental settings, we follow the original work by Walden *et al.* [8] using the *Random Forest* machine learning algorithm. We perform *within-project* evaluation, *i.e.*, training and testing data are obtained from the same project dataset. Recognizing that the dataset size can impact the performance of the VPM [39], we experiment with three data balancing techniques, as formerly done by Jimenez *et al.* [7]. Namely, we first use the dataset as-is, without any manipulation; then, we apply random *undersampling*, by which the samples in the majority class, *i.e.*, neutral files, are removed to match the minority class, *i.e.*, vulnerable files. Finally, we perform *oversampling* by employing the *Synthetic Minority Over-sampling TEchnique (SMOTE)* proposed by Chawla *et al.* [40], which consists in augmenting the number of samples in the minority class to match the size of the majority class, adding synthetic samples having similar features as the actual instances in the minority class.

Table 3 reports a summary of the experiments we run to answer our research questions.

3.3. Performance Evaluation

To evaluate the performance of the VPM, we rely on the confusion matrix, which summarizes the predictions made by the model. The number of True Positives (TP) is the count of source code files predicted to be vulnerable that are actually vulnerable; True Negatives (TN) are those files that were predicted to be neutral and are indeed neutral. False Positives (FP) indicate the number of files that the model labelled as vulnerable, but are actually neutral; False Negatives (FN) are those files predicted to be neutral, but that contain vulnerabilities. We take the following measures as indicators of the model’s performance:

- *Precision*. Indicates the percentage of actual vulnerable files among the ones predicted to

Evaluation Method	Dataset	Balancing technique	Number of experiments
Cross-validation	PHPMyAdmin	none	1
		undersampling	1
		oversampling	1
	Moodle	none	1
		undersampling	1
		oversampling	1
Release-based with perfect labelling	PHPMyAdmin	none	91
		undersampling	91
		oversampling	91
	Moodle	none	67
		undersampling	67
		oversampling	67
Release-based with real-world labelling	PHPMyAdmin	none	91
		undersampling	91
		oversampling	91
	Moodle	none	67
		undersampling	67
		oversampling	67
Total number of experiments			954

Table 3
Summary of the experiments executions.

be vulnerable and is given by the formula:

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

An high precision means that the model is often right when it classifies a file as vulnerable. The literature suggests [7, 10, 23, 39] that a value over 0.7 is reasonable.

- *Recall*. It is the percentage of files correctly labelled as vulnerable among all the actually vulnerable ones and indicates the ability of the model to recognize the vulnerable class.

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

- *Accuracy*. It is given by the formula:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- *Inspection rate*. It is the percentage of files labelled as vulnerable by the model and that therefore need to be inspected by developers to correct vulnerabilities. As defined by Walden et al. [8]:

$$InspectionRate = \frac{TP + FP}{TP + TN + FP + FN}$$

- *F1-score*. It is the harmonic mean of precision and recall:

$$F1 - score = 2 \cdot \frac{Recall \cdot Precision}{Recall + Precision}$$

- *Matthews Correlation Coefficient*. It is defined as a balanced measure of a classifier’s overall performance:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

4. Analysis of the Results

In the following, we report the main findings of our work, providing answers to the research questions driving our empirical study. In adherence to the principles of open science, we make all the data, scripts, and results of our study available in Appendix A.

4.1. RQ₁: Cross-validation

Our first research question pushed us to perform a replication of the original work by Walden *et al.* [8] to assess the performance of vulnerability prediction models in an *in-vitro* setting, *i.e.*, by executing cross-validation without taking into account the time relationships existing among data. Table 4 reports the performance of the considered VPM on the two datasets, manipulated with the three balancing strategies.

As expected, the results are encouraging, and consistent with the ones reported in the original work by Walden *et al.* [8]; namely, they experimented with the undersampled datasets. We observe that despite the dataset based on MOODLE includes fewer samples of vulnerable files, the performance of the VPM on it are comparable with the ones obtained on PHPMYADMIN. In fact, 0.7% of instances in the MOODLE dataset are labelled as vulnerable in the cross-validation setting, against 5.9% in the PHPMYADMIN dataset. This observation ignites the interest in knowing the minimum percentage of vulnerable files in a dataset needed to guarantee acceptable performance of VPMs; we aim at investigating this aspect in future work.

4.2. RQ₂: Release-based validation with perfect labelling

Our second research question was aimed at understanding the performance of VPMs in a partially-realistic setting, *i.e.*, when evaluated with a *release-based* approach. Figures 2 and 3 provides an overview of the performance of the model in such a setting, in which the perfect labelling assumption still holds.

When considering the dataset as-is, the performance obtained with the *release-based* validation method is quite good, even better than in the cross-validation scenario. We conjecture that this is due to the high imbalance of the datasets and the different sizes of the considered training and test sets. In fact, the highly unbalanced datasets contain a limited number of samples of vulnerable files, and plenty of samples of neutral ones. Hence, it is “easy” for the model to recognize neutral instances; this leads to a high number of True Negatives and a low number of

Dataset	Balancing technique	Precision	Recall	Accuracy	Inspection Rate	F1-score	MCC
PhpMyAdmin	none	0.98	0.78	0.99	0.05	0.84	0.85
	undersampling	0.97	0.82	0.89	0.42	0.89	0.81
	oversampling	0.99	0.99	0.99	0.56	0.98	0.98
Moodle	none	0.99	0.62	0.99	0.01	0.78	0.79
	undersampling	0.91	0.91	0.91	0.57	0.91	0.81
	oversampling	0.99	0.99	0.99	0.55	0.99	0.99

Table 4

Performance in the cross-validation setting, in terms of precision, recall, accuracy, inspection rate, f1-score and Matthew’s Correlation Coefficient.

False Negatives. The scarce presence of vulnerable samples in the datasets leads to low amounts of positive predictions in general. This has an impact on the computation of the performance indicators, *e.g.*, leading to high precision, recall and accuracy; therefore, the resulting overall performance seems quite promising. Nevertheless, we hypothesize that this is also due to the size of the training and test sets used to perform cross-validation and release-based validation, respectively. When applying cross-validation, the whole dataset is split into three folds, of which two are used for training, and the other one for testing. Therefore, the training-test data ratio is 2:1, and the size of the test set is similar to about 30 releases of PHPMYADMIN and about 25 releases of MOODLE. In the release-based validation approach, three releases are used for training and one is used for testing, thus the training-test data size ratio is 3:1, and the size of the test set is much lower. Given such observations, we conjecture that there are “fewer chances” for the model to make mistakes in the release-based validation setting, therefore the performance resulting from this evaluation method is quite better. However, we understand that the high imbalance of the dataset can affect all the performance measures in a substantial way, possibly invalidating our arguments.

When considering the undersampled datasets, we observe a considerable change in the performance; the release-based evaluation method leads to largely different performance, and we suppose that the cause has to be researched in the size of the training set. As explained before, in each round of the release-based validation, data belonging to three releases of the software are used to train the model, and we stress that the presence of vulnerable samples is rare. When applying undersampling, the number of neutral samples is reduced to match the number of vulnerable ones. Hence, the model ends up with low amounts of data to train, and cannot properly learn how to discriminate among the classes. This causes a severe decline in the performance. As the literature suggests that a precision value around 0.7 is reasonable [7, 10, 23, 39], such a model could not be employed in a real-case scenario, as the observed values for release-based validation vary between 0.2 and 0.4.

To generate the oversampled dataset, we followed Jimenez *et al.* [7] by using SMOTE to augment the number of vulnerable samples, *i.e.*, the minority class of the datasets. In general, the performance in such a scenario is better, as the model has “more opportunity” to learn in the training phase, since the dataset is larger and balanced. Still, the cross-validation setting

leads to better indicators, since the whole dataset is used; in the release-based validation only the data relative to three releases of the application is included in the training set.

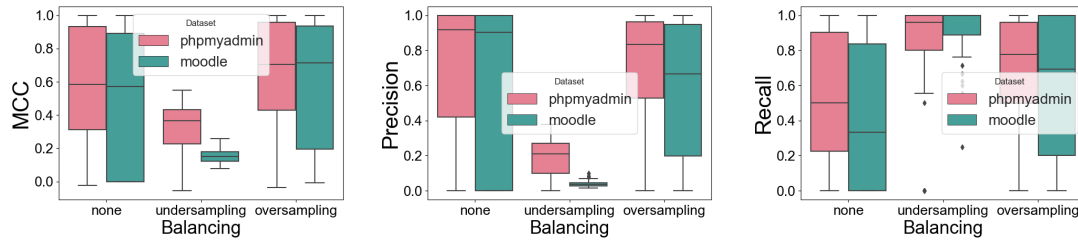


Figure 2: Performance in the release-based validation setting with perfect labelling, in terms of Matthew's Correlation Coefficient, precision, and recall.

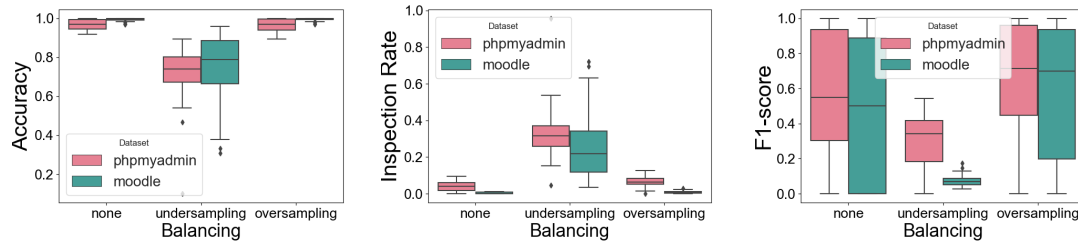


Figure 3: Performance in the release-based validation setting with perfect labelling, in terms of accuracy, inspection rate, and F-1 score.

4.3. RQ₃: Release-based validation with real-world labelling

Our third research question drove us to explore the practical applicability of vulnerability prediction models, evaluating them by employing a release-based approach and applying real-world labelling, *i.e.*, taking into account time relationships among data, we only leveraged the knowledge existing at training time to label the samples in the training and test set. Figures 4 and 5 show the performance of the VPM measured in the third experimental setting. The results reported in the box-and-whiskers plots are much more dispersed than what we observed in the context of RQ₂ and RQ₃, hinting at the fact that data which is not accurately crafted and is labelled with partial knowledge leads to oscillating performance.

In general, the performance are poorer than the ones observed in the other two evaluation scenarios. In particular, in a non-negligible number of cases, the model provides inadequate predictions. We conjecture that the degree of realism under which the evaluation is performed leads to lower performance. In fact, as vulnerabilities are discovered over time, it is not usual that security flaws are reported in the time span of a new software release, but much more time passes from the distribution of a product to the disclosure of a defect threatening it.

Therefore, the number of samples that are labelled as vulnerable according to the real-world labelling strategy is quite low with respect to the perfect labelling approach. This generates poor performance, as the model is not able to learn from small sets of samples. Surprisingly enough, the performance seem to get slightly better on undersampled datasets. We conjecture that this phenomenon is due to the model having “fewer chances” to make mistakes, as the number of instances to predict is lower. Still, negative MCC values are exhibited in the setting leveraging undersampled datasets; therefore, the model completely misunderstands the samples in many cases, predicting the opposite label.

Answering to RQ₃, we report that VPMs perform poorly when evaluated in a scenario which is similar to the actual context practitioners are supposed to employ them.

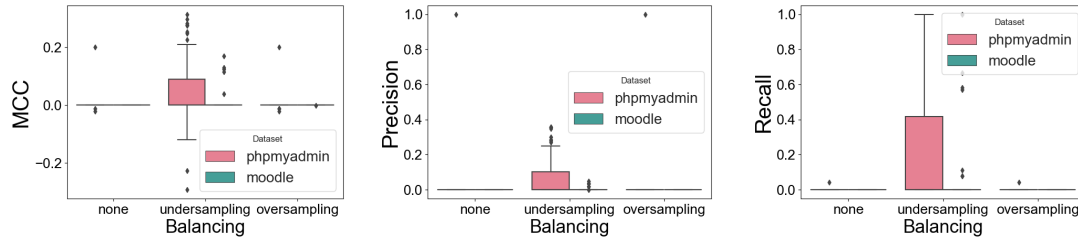


Figure 4: Performance in the release-based validation setting with real-world labelling, in terms of Matthew’s Correlation Coefficient, precision, and recall.

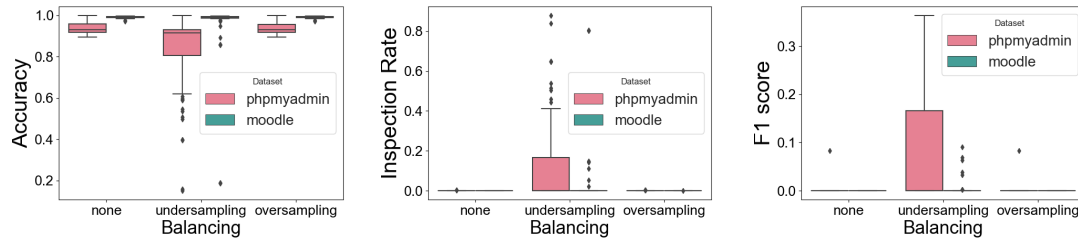



Figure 5: Performance in the release-based validation setting with real-world labelling, in terms of accuracy, inspection rate, and F-1 score.

5. Further Discussion and Take-Away Messages


To analyze the results of our empirical study, we observed the data coming from the executed experiments, deeply reasoning on the motivations behind the outcomes of the measurements. Such investigations led us to ignite further discussion points and distil a number of take-away messages that we believe can be relevant for the communities of researchers and practitioners.

Our first matter of disquisition arises from our deep investigation of the motivations behind the observed results, and from the work by Jimenez *et al.* [7] that called us to perform our

study. They examined the existing literature, and questioned the realistic usefulness of what was available in the landscape. We believe that all researchers are called to work similarly, not stopping at what the literature already provides, but always stimulating further reasoning, debating what is on the table, formulating new questions, and deeply studying the observed phenomena. This will make research continuously evolve, ultimately bearing fruit to the community of practitioners, and confirming the relevance of what we do.

 **Take-away Message 1.** Researchers should always push the advancement, not only by proposing novel work, but also by questioning, extending, and assessing the existing literature, in order to highlight potential weaknesses, understand, and overcome them.

Our study highlighted that the performance observed in *in-vitro* settings do not always resemble the true suitability of VPMs in realistic scenarios. We believe that practitioners should carefully select the proper approach to employ in their own software development process, according to their goals, context, and special needs. This raises the need for practitioners to leverage a tailored VPM, selected based on its complete set of characteristics.

 **Take-away Message 2.** There is the need for practitioners to be aware of the applicability and suitability of existing approaches in their own real software development processes.

6. Threats to Validity

In this section, we identify factors that may threaten the validity of our study, and present the actions we have taken to mitigate the risk, following the guidelines by Wohlin *et al.* [41].

External Validity. Threats to external validity are related to whether the observed experimental results can be generalized to other projects. For our experiments, we leveraged the dataset by Walden *et al.* [8] consisting in two popular open-source software projects written in PHP. We did not consider any other projects developed in other programming languages, such as C, C++, Python, Java, or any other kinds of software, such as desktop or mobile applications. Hence, we cannot claim that our results can be generalized to all software systems, as the projects under study may not be representative of software systems in general. Similarly, we cannot claim that our results can be generalized to a large set of vulnerability prediction models, since we experimented with a single one. To overcome this, extensive data coming from several projects written in different programming languages could be used for future experiments with additional kinds of models.

Internal Validity. Threats to internal validity are mainly concerned with the uncontrolled internal factors that might have influenced the experimental results. For the implementation of our experiments, we used third-party Python libraries, *i.e.*, SCIKIT-LEARN, and R, to avoid potential errors of a custom implementation of machine learning algorithms. We have made available the complete scripts and results; this allows the community to replicate our work and assess the validity of our findings.

Construct Validity. The construct validity relates to the suitability of the datasets and evaluation measures. It might be the case that the datasets used do not report all the security flaws

affecting the samples; this is understandable, as vulnerabilities are discovered over time, and software could be affected by threats that no one is yet aware of. For our experiments, we chose the well-known dataset by Walden *et al.* [8], that has been widely used and validated in the literature; therefore, we are confident that the data it provides is reliable. To create the ad-hoc datasets leveraged in the realistic setting, we leveraged the trustworthy information provided by the National Vulnerability Dataset and the Common Vulnerabilities and Exposures, which report detailed data describing the disclosed security flaws. To evaluate the performance of the VPM, we measured a set of largely used metrics that have been proposed and validated in the literature as good indicators of models' outcomes [5].

Conclusion Validity. Threats to conclusion validity impact the possibility to draw reliable conclusions. Comparing the performance of the VPM among the three evaluation settings is a non-trivial task, as the experiments are variegated. In our analyses, we conjectured a number of conclusions that might be supported by further research on the matter.

7. Conclusion

In this paper, we reacted to the alert raised by Jimenez *et al.* [7], who warned the research community working on vulnerability prediction models. They pointed out the importance of accounting for proper realism when evaluating VPMs to be used in the practice.

We performed an empirical study involving a well-known VPM [8] evaluated on two datasets manipulated with three data balancing techniques, executed at three different degrees of realism. First, we used cross-validation to evaluate the performance of the model, operating at *zero* realism. Afterwards, we exercised the model taking into account the time relationships existing among data, *i.e.*, applying a release-based evaluation approach [14, 23]. Finally, we operated in the fully-time-aware scenario by building a number of ad-hoc datasets only leveraging the vulnerability data that would be available at training time in practice. We found out that the performance of VPMs drop drastically when evaluated in a more realistic scenario, hinting that further research is needed to improve such models and make them useful for practitioners.

As a future part of our agenda, we want to extend our experiments by considering larger datasets to assess the reported findings. Furthermore, we plan to understand if and to what extent the training set size and the concentration of known vulnerabilities have an impact on the applicability of VPMs in practice. In particular, we expect that leveraging data collected throughout the whole history of a software project would be even more similar to what happens in real development scenarios, and perhaps beneficial for the performance. Finally, we want to investigate the employment of deep learning models to enhance the performance in the realistic scenario.

Acknowledgments

This work has been partially supported by (1) the EMELIOT national research project, which has been funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY), and (2) project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

References

- [1] M. Dowd, J. McDonald, J. Schuh, *The art of software security assessment: Identifying and preventing software vulnerabilities*, Pearson Education, 2006.
- [2] B. Liu, L. Shi, Z. Cai, M. Li, Software vulnerability discovery techniques: A survey, in: 2012 Fourth International Conference on Multimedia Information Networking and Security, 2012, pp. 152–156. doi:10.1109/MINES.2012.202.
- [3] M. Pistoia, S. Chandra, S. J. Fink, E. Yahav, A survey of static analysis methods for identifying security vulnerabilities in software systems, *IBM Systems Journal* 46 (2007) 265–288. doi:10.1147/sj.462.0265.
- [4] O. Zaazaa, H. El Bakkali, Dynamic vulnerability detection approaches and tools: State of the art, in: 2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS), 2020, pp. 1–6. doi:10.1109/ICDS50568.2020.9268686.
- [5] 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.
- [6] G. Lin, S. Wen, Q.-L. Han, J. Zhang, Y. Xiang, Software vulnerability detection using deep neural networks: A survey, *Proceedings of the IEEE* 108 (2020) 1825–1848. doi:10.1109/JPROC.2020.2993293.
- [7] 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: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019*, Association for Computing Machinery, New York, NY, USA, 2019, p. 695–705. URL: <https://doi.org/10.1145/3338906.3338941>. doi:10.1145/3338906.3338941.
- [8] J. Walden, J. Stuckman, R. Scandariato, Predicting vulnerable components: Software metrics vs text mining, in: 2014 IEEE 25th International Symposium on Software Reliability Engineering, 2014, pp. 23–33. doi:10.1109/ISSRE.2014.32.
- [9] V. R. V. R. Basili, G. Caldiera, H. D. Rombach, The goal question metric approach, *Encyclopedia of Software Engineering* (1994).
- [10] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 529–540.
- [11] A. Meneely, L. Williams, Secure open source collaboration: an empirical study of linus’ law, in: *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 453–462.
- [12] A. Meneely, L. Williams, Strengthening the empirical analysis of the relationship between linus’ law and software security, in: *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*, 2010, pp. 1–10.
- [13] E. Raymond, The cathedral and the bazaar, *Knowledge, Technology & Policy* 12 (1999) 23–49.
- [14] 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 (2010) 772–787.

- [15] T. Zimmermann, N. Nagappan, L. Williams, Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista, in: 2010 Third international conference on software testing, verification and validation, IEEE, 2010, pp. 421–428.
- [16] A. Hovsepyan, R. Scandariato, W. Joosen, J. Walden, Software vulnerability prediction using text analysis techniques, in: Proceedings of the 4th international workshop on Security measurements and metrics, 2012, pp. 7–10.
- [17] A. Kaya, A. S. Keceli, C. Catal, B. Tekinerdogan, The impact of feature types, classifiers, and data balancing techniques on software vulnerability prediction models, *Journal of Software: Evolution and Process* 31 (2019) e2164.
- [18] Q. Song, Y. Guo, M. Shepperd, A comprehensive investigation of the role of imbalanced learning for software defect prediction, *IEEE Transactions on Software Engineering* 45 (2018) 1253–1269.
- [19] X. Wu, W. Zheng, X. Xia, D. Lo, Data quality matters: A case study on data label correctness for security bug report prediction, *IEEE Transactions on Software Engineering* 48 (2021) 2541–2556.
- [20] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, S. Li, Combining software metrics and text features for vulnerable file prediction, in: 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE, 2015, pp. 40–49.
- [21] C. Catal, A. Akbulut, E. Ekenoglu, M. Alemdaroglu, Development of a software vulnerability prediction web service based on artificial neural networks, in: Trends and Applications in Knowledge Discovery and Data Mining, 2017, pp. 59 – 67. URL: https://link.springer.com/chapter/10.1007/978-3-319-67274-8_6.
- [22] P. Keller, A. K. Kaboré, L. Plein, J. Klein, Y. Le Traon, T. F. Bissyandé, What you see is what it means! semantic representation learning of code based on visualization and transfer learning, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31 (2021) 1–34.
- [23] Y. Shin, L. Williams, Can traditional fault prediction models be used for vulnerability prediction?, *Empirical Software Engineering* 18 (2013) 25–59.
- [24] F. Zhang, Q. Zheng, Y. Zou, A. E. Hassan, Cross-project defect prediction using a connectivity-based unsupervised classifier, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 309–320.
- [25] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, H. Leung, Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models, in: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, 2016, pp. 157–168.
- [26] J. Liu, Y. Zhou, Y. Yang, H. Lu, B. Xu, Code churn: A neglected metric in effort-aware just-in-time defect prediction, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2017, pp. 11–19.
- [27] M. Yan, Y. Fang, D. Lo, X. Xia, X. Zhang, File-level defect prediction: Unsupervised vs. supervised models, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2017, pp. 344–353.
- [28] X. Chen, Y. Zhao, Z. Cui, G. Meng, Y. Liu, Z. Wang, Large-scale empirical studies on effort-aware security vulnerability prediction methods, *IEEE Transactions on Reliability* 69 (2019) 70–87.

- [29] N. Li, M. Shepperd, Y. Guo, A systematic review of unsupervised learning techniques for software defect prediction, *Information and Software Technology* 122 (2020) 106287.
- [30] A. Garg, R. G. Degiovanni, M. Jimenez, M. Cordy, M. Papadakis, Y. Le Traon, Learning to predict vulnerabilities from vulnerability-fixes: A machine translation approach (2020).
- [31] R. Scandariato, J. Walden, Predicting vulnerable classes in an android application, in: *Proceedings of the 4th international workshop on Security measurements and metrics*, 2012, pp. 11–16.
- [32] R. Scandariato, J. Walden, A. Hovsepyan, W. Joosen, Predicting vulnerable software components via text mining, *IEEE Transactions on Software Engineering* 40 (2014) 993–1006.
- [33] M. Jimenez, M. Papadakis, Y. Le Traon, Vulnerability prediction models: A case study on the linux kernel, in: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2016, pp. 1–10.
- [34] G. Sellitto, F. Ferrucci, The impact of release-based training on software vulnerability prediction models, in: *8th ACM Celebration of Women in Computing (womENCourage)*, 2021.
- [35] A. A. Bangash, H. Sahar, A. Hindle, K. Ali, On the time-based conclusion stability of cross-project defect prediction models, *Empirical Software Engineering* 25 (2020) 5047–5083.
- [36] Q. Huang, X. Xia, D. Lo, Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction, in: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2017, pp. 159–170.
- [37] M. Tan, L. Tan, S. Dara, C. Mayeux, Online defect prediction for imbalanced data, in: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, IEEE, 2015, pp. 99–108.
- [38] J. Walden, J. Stuckman, R. Scandariato, Php security vulnerability dataset, in: 2014, 2014, pp. 23–33. URL: <https://seam.cs.umd.edu/webvuldata/>.
- [39] P. Morrison, K. Herzig, B. Murphy, L. Williams, Challenges with applying vulnerability prediction models, in: *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, HotSoS '15*, Association for Computing Machinery, New York, NY, USA, 2015. URL: <https://doi.org/10.1145/2746194.2746198>. doi:10.1145/2746194.2746198.
- [40] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, Smote: Synthetic minority over-sampling technique, *Journal of Artificial Intelligence Research* 16 (2002) 321–357. URL: <http://dx.doi.org/10.1613/jair.953>. doi:10.1613/jair.953.
- [41] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, *Experimentation in software engineering*, Springer Science & Business Media, 2012.

A. Replication Package

With the aim of adhering to the best practices of open science and reproducible research, we make available the complete data we leveraged in our work, along with the scripts we implemented to perform our analyses, and the comprehensive set of generated graphs and plots to visualize the results. The full replication package is available online: <https://doi.org/10.6084/m9.figshare.23574135>