The Importance of Accounting for Real-World Labelling When Predicting Software Vulnerabilities

Matthieu Jimenez
University of Luxembourg, Luxembourg
matthieu.jimenez@uni.lu

Renaud Rwemalika
University of Luxembourg, Luxembourg
renaud.rwemalika@uni.lu

Mike Papadakis
University of Luxembourg, Luxembourg
michail.papadakis@uni.lu

Federica Sarro
University College London, UK
f.sarro@ucl.ac.uk

Yves Le Traon
University of Luxembourg, Luxembourg
yves.letraon@uni.lu

Mark Harman
University College London and Facebook, UK
mark.harman@ucl.ac.uk

ABSTRACT

Previous work on vulnerability prediction assume that predictive models are trained with respect to perfect labelling information (includes labels from future, as yet undiscovered vulnerabilities). In this paper we present results from a comprehensive empirical study of 1,898 real-world vulnerabilities reported in 74 releases of three security-critical open source systems (`Linux Kernel`, `OpenSSL` and `Wiresark`). Our study investigates the effectiveness of three previously proposed vulnerability prediction approaches, in two settings: with and without the unrealistic labelling assumption. The results reveal that the unrealistic labelling assumption can profoundly mislead the scientific conclusions drawn; suggesting highly effective and deployable prediction results vanish when we fully account for realistically available labelling in the experimental methodology. More precisely, MCC mean values of predictive effectiveness drop from 0.77, 0.65 and 0.43 to 0.08, 0.22, 0.10 for `Linux Kernel`, `OpenSSL` and `Wiresark`, respectively. Similar results are also obtained for precision, recall and other assessments of predictive efficacy. The community therefore needs to upgrade experimental and empirical methodology for vulnerability prediction evaluation and development to ensure robust and actionable scientific findings.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**.

KEYWORDS

Software Vulnerabilities, Machine Learning, Prediction Modelling

ACM Reference Format:
Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. 2019. The Importance of Accounting for Real-World Labelling When Predicting Software Vulnerabilities. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338941

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5572-8/19/08.
https://doi.org/10.1145/3338906.3338941

1 INTRODUCTION

Manually assessing large-scale software systems for potential vulnerabilities is increasingly impractical, given that such systems may consist of many millions of lines of code, any of which might potentially contain vulnerability-inducing faults. Automated vulnerability prediction addresses this scalability challenge by adapting and augmenting widely-studied defect prediction techniques [28, 35, 37].

Vulnerability prediction systems use both software metrics (e.g., imports, function calls) and developer metrics (e.g., developers per component) which have also been used for defect prediction. The vulnerability prediction approaches previously proposed in the software engineering literature are each modified from traditional defect prediction to the more specific problem of vulnerability prediction by training on datasets that contain known vulnerabilities and using a variety of Machine Learning techniques to classify code as either vulnerable or not vulnerable.

Hitherto, empirical studies of the effectiveness of these vulnerability prediction approaches have implicitly assumed that labelling information is available regardless of temporal constraints (Table 1 summarises the evaluation method used to assess vulnerability prediction methods in previous work). That is, the methodology does not account for the gradual revelation of vulnerabilities over time; the vulnerability labels used for training the prediction models need to be more realistically available at training time and not include those subsequently uncovered.

New techniques for converting apparently non-vulnerable software faults into vulnerabilities are also discovered. Vulnerability detection is an adversarial process, in which those who seek to exploit faults continue to innovate.

The perfect labelling assumption that all vulnerabilities known from time t onwards are available at all times, even before t, is clearly unrealistic: a software engineer could only ever hope to train a predictive model on a *partial* ground truth that will include some degree of misclassification.

With the present state of the research literature on vulnerability prediction, we do not know the impact of this unrealistic perfect labelling assumption, because previous studies omit to discuss how they account for realistically available labelling at model training time. To address this issue we reformulate the methodology used to empirically evaluate vulnerability prediction. Our reformulated methodology takes full account of the vulnerability labelling information that could reasonably and realistically be assumed to be available to train any predictive model.

To realistically take account of the vulnerability information, we train the predictive models at time t, based on all (but only) those vulnerabilities known (already discovered) at time t, and then evaluate the so-trained model on its ability to predict vulnerabilities in subsequent releases.

We conducted a comprehensive empirical study of the three main previously proposed approaches, applied each one to guide the learning phase of a set of five widely-used machine learners, evaluated on a single large corpus of real vulnerabilities consisting of 1,898 vulnerable components and 74 releases from three open source systems. We study the effectiveness of these vulnerability prediction techniques in two scenarios, Firstly we train on the 'realistic scenario', by assuming that mislabelling noise is inevitable at any given time t, due to vulnerabilities unknown at time t. Secondly, we re-evaluate the vulnerability predictors with the same experimental settings, except that we make a more ideal 'perfect labelling' assumption: we assume that the training phase at time t has available to it, all labelling of vulnerabilities available at all times (even those that are discovered after time t).

Our findings provide strong evidence that the perfect labelling assumption is not only unrealistic, but that it can also mislead the scientific conclusions of any studies that make such an assumption. Therefore, future work on vulnerability prediction needs to use the reformulated methodology in order to ensure the scientific reliability of the conclusions drawn.

We have taken steps to ensure that this claim is based on firm foundations through the selection of a broad set of approaches, data sources, techniques considered and assumptions made. More specifically, our conclusions are based on a study of 1,898 vulnerabilities; an empirical evidence base that is approximately four times larger than any of the one previously used by the originally proposed methods [28, 35, 37]. The vulnerabilities used in our study are drawn from real-world systems, and concern previously reported real-world vulnerabilities, thereby avoiding any potential effects due to artificial or otherwise simulated systems or vulnerabilities. Our study also includes all the model features introduced in each and all of the previous studies.

Finally, in order to be as comprehensive as possible in our evaluation with respect to the knowledge realistically available at model training time, we also investigate a previously proposed, but as-yet-unevaluated, approach to improve predictive power. That is, we investigate the previous suggestion [38] to include *all* fault data available to the model training phase, including faults previously known to exist, yet not currently known to induce any vulnerability. The intuition for doing so is to exploit the potential link between bugs and vulnerabilities, i.e., bugs often provide indicators for vulnerabilities, even when these are currently not yet known, because no exploit has yet been found.

In summary the contributions of our paper are:

(1) We present the largest comprehensive empirical study on vulnerability prediction to date.
(2) We provide evidence that vulnerability prediction can be effective (MCC mean values of 0.77, 0.65 and 0.43 over the releases of `Linux`, `OpenSSL` and `Wiresark`, respectively) when making the *assumption* that perfect labelling is available to train the model.
(3) More importantly, we also show *dramatically lower* predictive effectiveness (MCC mean values of 0.08, 0.22 and 0.10 are achieved for `Linux`, `OpenSSL` and `Wiresark`, respectively) when we remove this unrealistic labelling assumption, instead training the models only on vulnerability labellings that could realistically be available to the learner at model training time.
(4) We investigate whether imbuing the training data with additional fault data from previously known faults that have not been determined to be vulnerabilities might enhance vulnerability prediction efficacy. These results show little improvement (MCC values are still below 0.30), and indicate that more work remains to be done to develop deployable vulnerability prediction for real world systems.

2 BACKGROUND

2.1 Security Vulnerabilities

A security vulnerability is defined as "*a mistake in software that can be directly used by a hacker to gain access to a system or network*" by the Common Vulnerability Exposures terminology [2]. Such mistakes are usually unexpected behaviours, backdoors, insufficient security measurements or code omissions (lack of defensive programming). Vulnerabilities are considered as of critical importance and their resolution is usually prioritized over other bugs. To this end, vendors usually make new releases in order to fix vulnerabilities faster and reduce their impact.

To support secure software products and vulnerability fixing, vulnerabilities are usually reported in publicly available databases. One such database is the National Vulnerability Database (NVD), which has been established by the National Institute of Standards and Technology (NIST) and U.S. government in order to encourage secure software development, public disclosure and management of vulnerabilities.

NVD is built upon the CVE List, which is a list of entries containing an identification number, a description and at least one public reference of the vulnerability. Thus, every publicly disclosed vulnerability is referenced with a unique identifier called Common Vulnerability Exposures (CVE) number or ID. NVD enriches each CVE entry with information such as the severity (named as CVSS) and the type (named as CWE) of a vulnerability. This data is continuously updated by the NVD staff [4].

2.2 Predictive Modelling for Software Security Vulnerabilities

Predictive modelling is a process of forecasting (future) outcomes (a.k.a., target or dependent variable) by using historical data. Each prediction model is composed by a number of predictors (a.k.a., independent variables) that are deemed likely to influence (predict) the future outcomes. Once historical data has been collected for relevant predictors, a prediction model can be generated using various techniques, such as statistical analysis, machine learning, or search-based algorithms.

Previous studies have shown that predictive modelling can be used to aid software engineers in their activities, ranging from project managment to software testing [12, 13, 16, 17, 30, 31, 33, 34].

In the context of software security vulnerabilities, predictive models have been used to classify part of the software as either predicted vulnerable or predicted non-vulnerable with the ultimate goal to support engineering in testing and code review activities.

For example, if one could identify with a high accuracy those part of the software that might be vulnerable engineers could prioritise their testing over testing other parts which are less likely to be vulnerable. Depending on the target analysis, prediction models may focus on different granularity levels, i.e., one can predict vulnerabilities at line, method, component or package level. In a sense the granularity level is the entity on the code based on which prioritization will be performed. Evidently, different granularity levels offer different advantages [27]. For instance, the line level granularity can be direct but can produce many false errors and can be too fine grained for the developers to identify issues. In our study we adopt the file-component granularity level following the findings of Morrison *et al.* [27], who found that the file (component) level was sufficient for Microsoft developers to work with. This decision is also in accordance to what most of the previously published work does [28, 35, 37].

Once established the granularity level, the dependent variable should indicate whether a target component contains one or more vulnerabilities, while the independent variables (i.e. predictors) can be many and related to different aspects of the software and its production. In the literature, three main methods have been proposed to extract vulnerability predictors from historical data, named as *Imports and Function Calls* [28], *Code and Process Metrics* [37, 38], and *Bag of Words* [35]. These predictors can be used with traditional machine learning classifiers in order to build prediction models. In our study we realise and investigate all the three methods to extract different predictors set as to the best of our knowledge there has been no study comparing these approaches on a level playing field so far, and assess their effectiveness in combination with five different machine learners (i.e., AdaBoost, J48, K-Nearest Neighbourhood, Logistic Regression and Random Forest).

2.3 Methods to Extract Vulnerability Prediction Features

Here we describe the three main methods proposed in literature to extract from historical data the vulnerabilities predictors that can be used as input to automated vulnerability prediction systems.

Imports and Function Calls

Neuhaus *et al.* [28] observed that vulnerable files tend to import and call a particular small set of functions. Based on this observation they suggested the first approach that implements vulnerability prediction. This is a simple prediction model over the components' imports and function calls. In other words, the imports and function calls are the training features.

To apply this prediction modelling technique one needs to extract the imports and function calls of the components under analysis. In our experiment, we retrieve this information by traversing the Abstract Syntax Trees (ASTs) of the files. Following the recommendation of Neuhaus *et al.* we use imports and function calls as a separate set of features and, therefore, trained two models (one per each set).

Code and Process Metrics

Shin *et al.* [37, 38] used code metrics related to code complexity, code churn and developer activity to build vulnerability prediction. According to these studies, the combined use of these metrics gives the best results. In summary the features used by this approach are the following:

- `Complexity and Coupling:`
  LinesOfCode: lines of code;
  PreprocessorLines: preprocessing lines of code;
  CommentDensity ratio: lines of comments to lines of code;
  CountDeclFunction: number of functions defined;
  CountDeclVariable: number of variables defined;
  CC(sum, avg, max): sum, average and max cyclomatic complexity;
  SCC(sum, avg, max): strict cyclomatic complexity [37];
  CCE(sum, avg, max): essential cyclomatic complexity [37];
  MaxNesting(sum, avg, max): maximum nesting level of control constructs;
  fanIn(sum, avg, max): number of inputs, i.e., input parameters and global variables to functions;
  fanOut(sum, avg, max): number of outputs, i.e., assignments to global variables and parameters of function calls.

- `Code Churn:` *added lines*, *modified lines* and *deleted lines* in the history of a component.

- `Developer Activity Metrics:`
  number of commits impacting a component;
  number of developers modified a component;
  current number of developers working on a component.

We computed the above metrics by analysing the program AST and the Git history.

Bag of Words

This approach treats code as a set of words. It tokenizes the code and puts every token into a reference bag along with its appearance frequency. It is known as text mining and has been suggested for vulnerability prediction [35]. The features are the appearance frequency of the tokens, i.e., unigrams, in the code of the components. As the features dimensionality explodes quickly reducing it is mandatory [35, 40]. To this end, previous studies [35, 40] discretized the frequency of tokens (to make them binary) using the method of Kononenko [24].

3 RELATED WORK

Vulnerability prediction has been attempted in previous studies differing from each other mainly from the predictors used, the subject and the validation carried out. In this paper, we investigate all previously proposed predictors' sets (creating a different prediction model using each for each set as described in Section 2.3 and validate them as done in previous work and also in a more realistic scenario. Table 1 summarises and compares the key aspects of related work with respect to our work, including their validation procedure and whether they consider mislabelling noise. We can observe that all studies perform a cross-validation, which does not consider the temporal aspect and therefore it is unrealistic.

Table 1: Comparison with previous work. All studies perform cross-validation to assess the effectiveness of the prediction models and only three of them also consider a release-based validation. Cross-validation does not consider temporal aspects and therefore it is unrealistic. Among the three release-based studies, one uses synthetic data, one does not consider mislabelling noise and the other does not specify.

Study	Systems	No. of Vulnerabilities	Granularity	Method	Evaluation Method	Results	Mislabelling Noise
Neuhaus et al. [28]	`Mozilla`	134	Component	Imports and Function calls	Cross-validation	Precision 70, Recall 45	Does not consider
Zimmermann et al. [46]	Windows Vista	66	Binary	Code Metrics & Dependencies	Cross-validation	Precision 66.7 & 60, Recall 20 & 40	Not specified
Shin et al. [37]	`Mozilla Firefox`, `Red Hat Enterprise`, `Linux kernel`	389	File	Code Metrics	Cross-validation & Release-based	Precision 3 - 5 & 3, Recall 87 - 90 & 79 - 85	Not specified
Shin et al. [38]	`Mozilla Firefox`	363*	File	Code Metrics	Cross-validation	Precision 9, Recall 91	Does not consider
Scandariato et al. [35]	20 `Android` apps	N/A	File	Bag Of Words	Cross-validation & Release-based	Precision 90** & 86**, Recall 77** & 77**	Does not consider and uses artificial data
Walden et al. [43]	`Drupal`, `Moodle`, `PHPMyAdmin`	223	File	Bag Of Words	Cross-validation	Precision 2-57, Recall 74-81	Does not consider
Zhang et al. [45]	`Drupal`, `Moodle`, `PHPMyAdmin`	223	File	Code Metrics & Text Features	Cross-validation	Precision 25-67, Recall 4-69	Does not consider
Jimenez et al. [22]	`Linux Kernel`	743	File	Imports and Function calls, Code Metrics, Bag Of Words	Cross-validation & Release-based	Precision 65-76 & 39-93 , Recall 22-64 & 16-48	Does not consider
This paper (“Ideal” World) (“Real” World)	`Linux Kernel`, `OpenSSL`, `Wireshark`	**1593**	**File**	**Imports and Function calls, Code Metrics, Bag Of Words**	**Release-based**	**Precision 44.7-83.3 , Recall 33.5-76.5 Precision 19.4-44.6, Recall 1.6-26.2**	**Evaluates the impact of mislabelling noise**

*Number of vulnerable files. **Estimated from the graphs and reported data of the paper.

On the other end, three of these studies perform also a release-based analysis but one uses synthetic data, one does not consider mislabelling noise and the other does not specify. Therefore we can conclude that all previous studies overlooked temporal labelling assumptions, and our paper is the first to analyse the impact of this assumption on the quality of the prediction. Indeed, our “Ideal” world results are close to those reported by previous work ignoring time when labelling, however when the same models are used in a realistic scenario their predictive performance dramatically drop revealing that previously reported results were optimistic. In the following we describe each of the previous work in detail.

Neuhaus *et al.* [28] were the first to find a correlation between import/function calls and vulnerabilities and to use the import and function calls as features to train a classifier able to predict vulnerable components. They empirically evaluated this proposal for the Mozilla Firefox project achieving a recall of 45% and a precision of 70%.

Shin *et al.* [37] experimented with complexity metrics along with code churn and developer metrics. The authors validated their approach for Mozilla Firefox and Red Hat Linux and obtained a recall of up to 86% for Mozilla Firefox and up to 90% for the Linux. However, they reported a low precision. Subsequently, the same authors analysed whether a traditional defect prediction models, trained on complexity, code churn and past fault history is capable of predicting software vulnerabilities [38]. They found that distinguish between bugs and vulnerabilities is a hard task, as they obtained similar results for both cases.

Chowdhury and Zulkernine [9, 10] proposed a similar approach but using a slightly different set of metrics: complexity, coupling and cohesion. The evaluation performed for Mozilla Firefox showed an average recall of 74.22%.

Zimmerman *et al.* [46] carried out an empirical study to evaluate the efficacy of code churn, code complexity, dependencies and organizational measures to build a vulnerability prediction model for Windows Vista. Their proposal obtained a good precision but low recall. Nguyen *et al.* [29] used an approach based on dependency graph, rather than traditional source code metrics, to train the vulnerabilities prediction model and evaluated it on Mozilla Javascript Engine obtaining an average precision and recall of 60%.

Recently, Scandariato *et al.* [35] investigated the use of text mining. The combination of natural language processing and prediction models was introduced for defect prediction by Hata *et al.* [19] and has been successfully used for other software engineering prediction tasks [14][31]. Scandariato *et al.* [35] decompose the source code into a bag of words which is then used to train a classifier. This approach was validated for 20 android applications, yielding a precision and recall of about 80%. However, the dataset used in this study was built using a static analysis tool and since these tools are quite imprecise they might produce a lot of type I and type II errors. Such concerns were addressed by Walden *et al.* [43] who evaluated the same approach on different settings. They used a dataset composed by three web applications written in PHP, for a total of about 30 vulnerabilities per application and applied cross-validation as there was not enough data to create two independent sets. This undermine the validity of the results since the evaluation settings used have been shown to lead to generalization and overfitting problems [11]. Using the same dataset, Zhang *et al.* [45] propose to combine code metrics and text mining techniques. Overall, the authors manage to improve the results for precision while the recall is only improved in one case.

Jimenez *et al.* [22] carried out an empirical study comparing the vulnerability prediction approaches using a dataset of 743 vulnerabilities from the `Linux Kernel` (which was split into independent training and evaluation data sets) and found that function calls and text mining were the best performing approaches. Although related, Jimenez *et al.* used a commit-based analysis for only one system (while herein we use a release-based one for three systems) and does not investigate the impact of data leakage.

The approaches discussed so far are somehow generic and can work with most of the existing software. However, they do not specialise on specific types of vulnerabilities. Two examples of approaches requiring additional data and/or the help of a tool are those of Smith *et al.* [39] and Theisen *et al.* [42]. Smith *et al.* [39] approach for SQL Hotspot revealed a correlation between vulnerabilities and the number of SQL statements. Theisen *et al.* [42] suggested to use crash dumps to identify part of system that might be vulnerable. In particular, they define the notion of attack surface approximation that can be used to help vulnerability prediction. The authors empirically compared a model based on this approach against one based on code metrics using a Windows 8 vulnerability dataset and found slightly better results based on attack surface approximation. Since this approach is unsupervised it does not require labelling the training data, which is an advantage, however recall cannot be achieved as there is no information available for all those source code files that do not have a crash history.

4 RESEARCH QUESTIONS

We start our empirical study by assessing the effectiveness of previously proposed vulnerability prediction techniques in the realistic setting, in which only reasonably available vulnerability labelling are assumed to be available at model training time.

RQ1: *How well do prediction models identify vulnerable components between software releases in the 'real world', using the reformulated (and more robust) experimental methodology?*

To establish realistic settings in answering RQ1 we train the prediction models using the information available (reported vulnerabilities) at release time and evaluate against the 'ground truth' data (vulnerabilities reported over the whole period of time that we consider).

After checking the performance of prediction models in this realistic setting we repeat the entire process for the 'ideal world' setting, in which all vulnerability labels (known at any time) are also assumed to be all available at model training times.

RQ2: *How well do prediction models identify vulnerable components between software releases in the "ideal" world?*

Finally, we wish to be comprehensive, so we also evaluate the suggestion, raised in previous work [38], that previously discovered faults (not known to be vulnerabilities) should be included in the training data available since this might improve vulnerability prediction. Hence we ask:

RQ3: *Can we improve the accuracy of vulnerability prediction in the 'real world' setting by providing prediction models with more general defect-based information?*

To answer RQ3, we repeat the analysis carried out for RQ1 but we use modified training sets, and compare their performance against the models built using the original training sets. In particular, we include in the training sets defect-related information. We thus, augment the training sets with components which were defective but considering them as vulnerable by assigning them a lower weight (equal to one) with respect to the weight (equal to five) of the actual vulnerable components. This practise is known as *training set augmentation* and attempts to tackle the insufficient learning signal and the class imbalance problem [38].

Table 2: Vulnerability data in our corpus.

Software System	Vulnerabilities	Vulnerable Components
`Linux Kernel`	1,202	1,508
`Wireshark`	265	221
`OpenSSL`	126	164
Total	**1,593**	**1,898**

5 CORPUS

In our study we consider three large security intensive open-source software systems: the `Linux Kernel`, the `OpenSSL` library and the `Wireshark` tool. These systems are widely-used, mature and have a long history of releases and vulnerability reports, which is needed to perform realistic experiments with machine learning. Additionally, these systems are publicly available on Git, which allows for their releases analysis by simply linking them with the NVD, moreover other researchers can access the same data for reproducibility and extensions. In the following we describe these systems, the procedure we followed to collect the data and the characteristics of the vulnerabilities we collected. Additional details about the data collection and analysis can be found in the dissertation of Matthieu Jimenez [21].

5.1 Software Systems

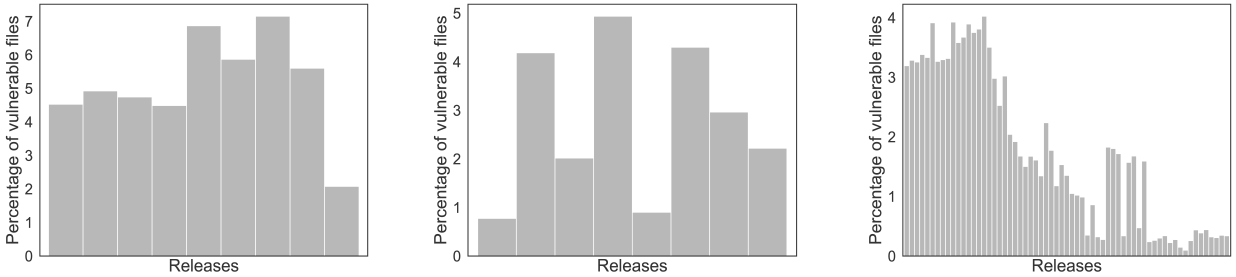
The `Linux Kernel` is an operating system. To date it is integrated in billions of systems and devices such as Android. Linux is one of the largest open-source code-bases including approximately 19.5 million LOC and has a long history (since 1991), recorded in its repository. It is relevant for our study as it has many security aspects and is among the projects with the higher number of reported vulnerabilities in NVD.

`OpenSSL` is a library implementing the SSL and TLS protocols, commonly used in communications. In 2014 the project was used by more than 65% of the web servers worldwide [1]. OpenSSL has approximately 650 KLOC. It is relevant for our study because of its critical importance, as highlighted by the heartbleed vulnerability, which made half of a million web servers vulnerable to attacks [3].

`Wireshark` is a network packet analyser mainly used for troubleshooting and debugging. It supports developers and network managers by capturing traffic, analysis protocol and interface controller behaviour. The project is open source and involves more 3.6 million LOC and is relevant for our study because it is integrated on most operating systems.

5.2 Data Collection

We collected all vulnerabilities reported in NVD for the three systems under study using the VulData7 framework [23]. VulData7 automatically retrieves all declared bug reports and patches by crawling NVD. Using this information the framework retrieves all the related commits from Git, i.e., for each vulnerability that has a link to a patch. To make sure that VulData7 retrieves all reported vulnerabilities it also searches the projects version history to identify commit messages with references to CVEs or bug IDs mentioned by the NVD data.



(a) OpenSSL (b) Wireshark (c) Linux Kernel

Figure 1: Ratio of vulnerable components per considered release in the three systems we study. We consider 64 versions of `Linux Kernel`, (versions from 2.6.12 to 4.15), 9 versions of `Wireshark` (all major releases) and 10 versions of `OpenSSL` (all major releases).

The above process provides our data. We believe that our data are relatively precise in the sense that components tagged as vulnerable are indeed vulnerable. This is because developers acknowledge the existence of vulnerabilities and independent practitioners validated the reports [27, 46]. We further discuss this issue in the Threats to Validity Section 6.4.

We consider a component as vulnerable if it was modified in order to fix the vulnerability. This is in line with previous work, which considers the components that were blamed by security reports, e.g., Neuhaus *et al.* [28] and Shin *et al.* [37]. Table 2 reports a summary of our collected data. Of course, while we can be relatively sure that code labelled as 'vulnerable' does, indeed, expose a vulnerability, we cannot be so certain of the absence of vulnerabilities in code marked as 'non-vulnerable'.

Since we investigate vulnerabilities per release, we use data from previous release(s) to predict the vulnerable components of the next release. To characterize such data, we mapped the vulnerabilities with the releases they affected (using the NVD data). The NVD dataset tells us which releases are affected by a vulnerability and we further mine vulnerability fixes which means we know when the vulnerabilities were removed. These two observations give us a reasonably reliable assessment of the lifetime of a vulnerability.

Our goal is to perform vulnerability prediction between software releases. This means that at a given release, we want to predict the vulnerabilities in the next release. However, at the given release time, there might be vulnerable components that have not yet been reported. As we already discussed, this fact introduces a significant challenge to the approach since it provides wrong learning signals, i.e., vulnerable components (unreported vulnerabilities) are considered as clean/non-vulnerable. To account for this we form two datasets, one containing all vulnerabilities reported at every examined release time (corresponding to a 'realistic' scenario - investigated by RQ1), and one containing all vulnerabilities irrespective to when they were reported (corresponding to an 'experimental' scenario similar to the one followed by previous work - investigated by RQ2). Thus, we have the following two datasets:

Realistic Data - used in RQ1 & RQ3: *The historical data (project components) is labelled as vulnerable and clean (i.e. non-vulnerable) according to the information available at release time (i.e. vulnerabilities reported before the particular release).*

Experimental Data - used in RQ2: *The historical data (project components) are labelled as vulnerable and clean according to our "ground truth" data (vulnerabilities reported over the whole period of time that we consider).*

Using VulData7 and by following the above data collections process we gathered a set of vulnerable and clean components falling into 64 versions of `Linux Kernel` (from 2.6.12 to 4.15), 9 versions of `Wireshark` (all major releases, i.e., ending with 0) and 10 versions of `OpenSSL` (all major releases).

Figure 1 shows the ratio of vulnerable components per release for the three systems we study. The data show that the proportion of vulnerable components ranges from 1% to 7% for all the systems we studied. Such a ratio indicates a largely unbalanced dataset, which can be challenging for the prediction modelling methods [27]. We also observe that the number of vulnerable components rises up each major releases.

6 EXPERIMENTAL DESIGN AND ANALYSIS

6.1 Methodology

To evaluate vulnerability prediction and answer our RQs we performed the following analysis. For every considered release, we iteratively train on the previous release(s) and evaluate on the current one. We consider two typical cases addressed in previous work: training on the last release [18, 20] and training on the last three releases [37, 38]. We start the evaluation from the fourth release onwards (as we need at least three releases on which to train the predictive models) and we consider releases with at least 10 vulnerable components. These constraints ensures that we have sufficient data for analysis. As a result of this procedure we evaluate on 61, 6 and 7 releases of `Linux Kernek`, `OpenSSL` and `Wireshark`, respectively.

Since our data is imbalanced (less than 10% vulnerable components per release compared to the non-vulnerable ones), we investigate the use of the Synthetic Minority Over-sampling TEchnique (SMOTE) [8], which augments the minority class data with synthesised instances that have similar feature values with the real ones (of the minority class). We repeat all our experiments with and without SMOTE. To avoid bias from a specific classifier[1] we considered five popular ones, i.e., AdaBoost, J48, K-Nearest Neighbourhood (k=5), Logistic Regression and Random Forest. These classifiers are typically used in prediction studies. To train and evaluate the models we used WEKA [5] version 3.9.2, which is publicly available, thus allowing for the reproducibility of our study.

To train and test, a feature matrix needs to be constructed. In this matrix, the columns correspond to the different features and the rows to the components (values of the features for each component). We used and evaluate three different features set produced by the methods. In *Code Metrics*, the feature matrix has a fixed number of columns, i.e., one per metric. However, the *Imports*, *Function Call* and the *Bag of Words* have a non constant number of columns (these are determined by the features extracted from the training set). This is likely to introduce a scalability challenge due to the large number of features. To reduce this burdern we used 'minimum support' [28], i.e., a minimum amount of cases a potential feature needs to appear in the training set, to be considered as a feature, and set it to 5%.

6.2 Performance Measurement

Vulnerability prediction is treated as a binary classification (predicting components as being vulnerable and non-vulnerable). Thus, predictions might characterize components as: vulnerable while they are not (False Positives - FPs), non-vulnerable while they are vulnerable (False Negatives - FNs), vulnerable while they are (True Positives - TPs) and non-vulnerable while they are (True Negatives - TNs). A typical way to evaluate such methods is by using Precision and Recall measures. These are defined as:

$$\textbf{Precision} = TP \ / \ (TP + FP)$$

$$\textbf{Recall} = TP \ / \ (TP + FN)$$

Traditional information retrieval metrics such as recall, precision, F-measure, and ROC-AUC do not take into account the "true negative" count and can be misleading, especially when using imbalanced data. Therefore, we complement their use with the Matthews Correlation Coefficient (MCC) [26], a reliable metric of the quality of prediction models [7, 36]. This metric takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even when the classes are of very different sizes, as in our case. The *MCC* is defined as:

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

[1]It is unknown which classifier performs best for a given problem. Therefore, we need to check this before performing any prediction modelling evaluation.

As such, *MCC* returns a coefficient between 1 and -1, where $MCC = 1$ indicates a perfect prediction, $MCC = -1$ a perfect inverse prediction (i.e., total disagreement between prediction and observation) and $MCC = 0$ indicates that the classifier performance is equivalent to random guessing.

Generally, classifiers are not directly returning a class in which an element is supposedly belonging, but are instead returning a probability of the said element to belong to a class. This probability can then be used to classify the element into one of the classes, depending on a threshold. This probability of a component to be vulnerable according to a model, that we call *Prediction probability PredP*, can be used as a ranking basis where component would be ranked in ascending order. The position of a specific component in this ranking gives us its *Relevant Ranking* (*RR*).

We also consider the effort put by the engineers to use the classification produced, i.e., the number of components that need inspection in order for someone to inspect all vulnerable components. This metric is defined as:

$$\textbf{Resolution effort ratio} = \frac{Number\ of\ components\ needing\ inspection}{Number\ of\ components}$$

As engineers may focus on the most likely cases, we also need a metric focussing on the top ranked components. We consider the top-*n* metric, which is defined as the number of vulnerable components in the top *n* places of the *RR*. We use *n* value of 10.

6.3 Pre-analysis

Our experiment involves a large number of settings that require the repetition of our analysis 16,760 times, as we need to investigate 4 approaches with 5 classifiers, 2 validation ways (using one release or the the last three), 2 ways of handling imbalance (using/not using SMOTE) for the 3 systems of interest (74 releases studied in total) and for 3 different RQs. To deal with this issue we first determined the best combination of the pair <classifier, prediction method> for all the releases of our corpus and then we applied the best-performing ones to answer our RQs.

In particular, we gather the results of RQ1 for all combination and ranked the performance of the classifiers for a given approach. Then, we computed the average rank of all pairs. Detailed results are omitted due to lack of space. The best classifier is Random Forest (it achieves the best results in all cases). Therefore, we used this pair in our analysis.

We then evaluate whether we should use SMOTE or not and whether we should train on the last three releases or just on the last one. We thus, computed all the MCC values and compared them using the Wilcoxon signed rank test [44]. We found that SMOTE does have a positive effect, with statistically significantly better performance metrics for all the approaches and classifiers' combinations we examined ($p - value < 0.005$). We also found no statistically significant differences between using the last release and using the last three releases for training the prediction models. Though, we decided to perform our analysis using the three last releases as it provided slightly more stable results.

In conclusion, in the rest of our analysis we use the following settings: training using SMOTE on the last three releases with the Random Forest classifier.

6.4 Threats to Validity

A potential threat to our study may arise from data collection. We work with publicly reported vulnerabilities mined from code repositories. This process ensures the retrieval of components that developers mark as vulnerable and the affected releases (reported in NVD) but tells nothing about the unmarked components (which we consider as non-vulnerable). Yet, in practice this might not be the case and may result in false negatives and false positives that reduce performance, as we show in RQ1. To reduce this threat we selected mature projects with a long history of releases and vulnerability reports. We also assumed that training is performed on the publicly reported vulnerabilities. Yet, in practice practitioners may have additional information available (vulnerabilities they found and did not report), which can boost the performance that we observe by comparing the results of RQ1 with those of RQ2.

Potential defects in our framework may unintentionally influence our results. To reduce this threat, we carefully inspected our code and tested it. To further reduce it and enable replication, we make our data publicly available[2]. Threats may also arise due to the classifiers' configurations [6]. All classifiers were configured with the hyper-parameters setting of WEKA: altough using a same default setting allowed us to compare all the techniques on a level playing field, performing hyper-parameter tuning could further improve performance [15, 25, 32, 41].

Also, we did not use the SVM classifier as in the study of Neuhaus *et al.* [28] due to technical problems with WEKA when using it on the bag of words and metrics methods. Nevertheless, we achieved better results with Random Forest than with the Support Vector Machines for the particular case of Neuhaus *et al.* Additionally, previous research on bag of words [35] and metrics [37] showed that Random Forest and Linear Regression performed better than Support Vector Machines.

Finally, we cannot claim that our results generalise beyond the subjects studied. However, to reduce this threat, we studied three large open-source systems with a large number (3-4 times larger than in previous studies) of real (reported in NVD) vulnerabilities. Moreover, we use a publicly available tool (VulData7 [23]) for data collection and report the followed procedure in order to allow other researcher to replicate and extend our work.

7 RESULTS

7.1 RQ1: Performance in the "Real" World Setting

This RQ investigates the practical applicability of the prediction models using a realistic labeling taking into account temporal constraints, i.e., we investigate the predictions that one can achieve by using only those vulnerabilities that were reported at the software release time .

Figure 2 shows the distribution of the predictions' evaluation measures (MCC, Precision and Recall) for all methods and subjects under studies. We can observe little differences between the approaches, in terms of the MCC values, only *Code Metrics* performs slightly better (measured by MCC) than the other three methods for `Linux Kernel`.

In terms of precision and recall *Code Metrics* performs better than the other methods for `OpenSSL`, slightly worst for `Wiresark` and has almost identical behaviour to the other methods for `Linux Kernel`. These results show that the choice of the vulnerability predictors does not noticeably impact the prediction performance.

Moreover, we can observe that all predictions are consistently poor: all performance measures are on average lower than 0.5. Literature suggests (e.g., [28, 38]) that precision and recall values ≥ 0.7 are reasonable. An industrial study at Microsoft [27] suggests that "False positive rates around 0.5 would still not be good enough to trigger real action". Therefore, according to these thresholds, the studied models achieve non-actionable results when used in a realistic setting. This is an important finding and surprising given previously reported results, which were based on an unrealistic temporal labeling assumption [22, 28, 35, 37, 38, 43, 45, 46].

Figure 5(a) shows the results with respect to resolution effort ratio and top-10 metrics. The effort ratio results represent the required inspection effort to identify all vulnerable components. Unfortunately, the results demonstrate that engineers will need to inspect almost the whole codebase, indicating a rather poor performance. The top-10 results are in line with the precision results found earlier, i.e., they show that engineers can identify 2 to 5 vulnerable components among the top 10 suggested.

Therefore, our answer to RQ1 is that in a realistic setting these approaches have poor performance (lower than 0.4 precision and 0.2 recall on average).

7.2 RQ2: Performance in the "Ideal" World Setting

Figure 3 shows the distribution of the predictions' evaluation measures for all methods and subjects we consider. The results are consistent for all the studied subjects, with the *Bag of Words* and *Function Calls* methods achieving better results than other in terms of MCC, while smaller differences are observed in terms of precision and recall.

Interestingly, in this case we observe very good results for the `Linux Kernel` subject with median values of MCC, Precision and Recall above 0.8 for all approaches (except the recall of *CodeMetrics*). The results for `OpenSSL` are lower than those of `Linux Kernel`, but still good overall (above 0.50 for all measures). The results for `Wireshark` are much variable with Recall and Precision values ranging from 0.20 to 0.70 (mainly due to the low prevalence of vulnerabilities in the project releases of `Wireshark`). The resolution effort ratio and top-10 measures shown in 5(b) confirm the good performance of these models: Engineers need to inspect a smaller portion of the codebase to find all vulnerabilities and can identify 7 to 10 vulnerable components among the top 10 suggested in `OpenSSL` and `Linux`, while 4 to 6 in `Wireshark`. Although these results seem very good, they are optimistic as RQ1 showed that using a more realistic labeling they dramatically decrease.

Therefore, even if the "ideal" world vulnerability prediction achieves results that according to the literature can be characterized as actionable [27, 28, 38], the gap with "real" world is too large, indicating that the vulnerability prediction techniques proposed so far are not yet practically applicable.

[2] https://github.com/kabinja/fse2019

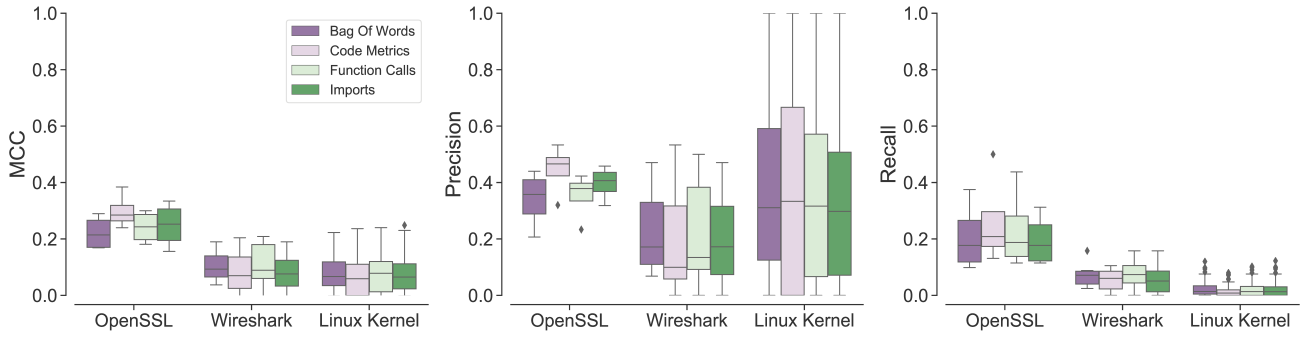


Figure 2: Prediction performance in a "real" world setting - with data labelled only with information available at model-building time(RQ1). We observe a relatively low performance for all methods. Also, all approaches show little performance differences (all yield similar MCC values).

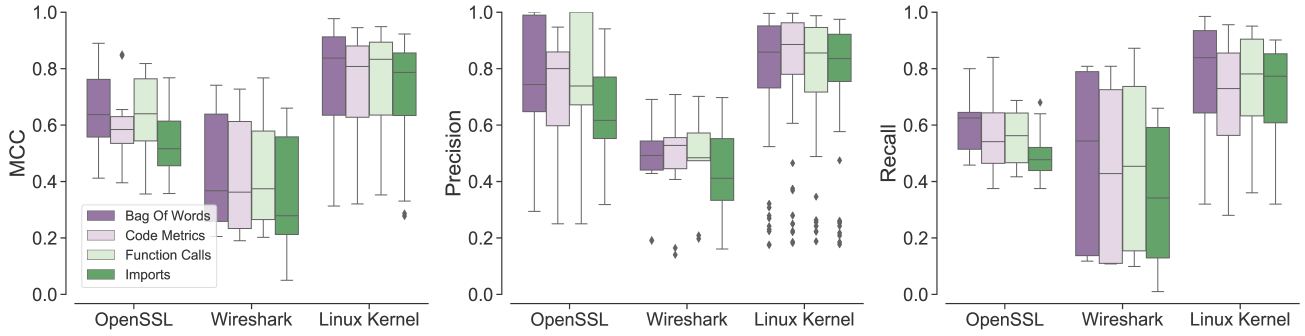


Figure 3: Prediction performance in an "ideal" world setting - with unrealistic vulnerability labelling information (RQ2). We observe a relatively good performance for all methods. *Bag of Words* and *Function Calls* methods achieve the best results, i.e., yield the highest MCC, but overall with negligible differences. There are notable performance differences from the "real" world results presented in Figure 2.

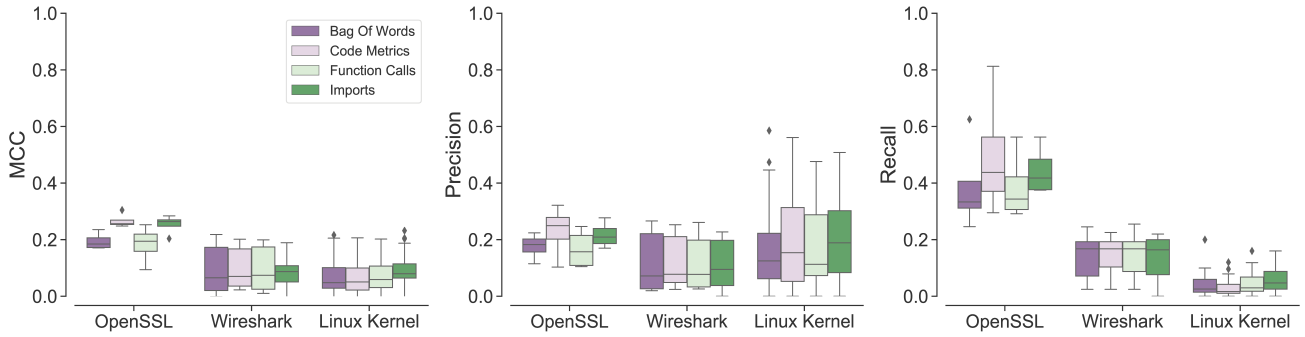


Figure 4: Performances for training set augmentation in a "real" world setting (RQ3). We observe that training set augmentation does not noticeably help achieving a reasonable performance. There are no significant performance differences from the non-augmented results that were presented in Figure 2.

7.3 RQ3: Performance in the "Real" World Setting with Augmented Training Data

Figure 4 shows the performance of vulnerability prediction when augmenting training data to improve predictions, as suggested by Shin *et al.* [38]. The figure presents the distributions of MCC, Precision and Recall values. We observe that training set augmentation does not improve much the overall performance as the MCC values are relatively low. In fact by comparing the results of Figures 2 and 4 we can see that training set augmentation does not improve performance (in fact it has a negative effect) in terms of MCC. Interestingly, we also observe that training set augmentation results in trading recall over precision. By comparing the results of Figures 2 and 4 with respect to Recall and Precision values we see that Recall is increased while Precision is decreased. This can be explained by the fact that only in the 'bug' strategy we are altering the learning phase by adding examples.

Figure 5(c) presents the results with respect to resolution effort ratio and top-10 metrics. The effort ratio results indicate that, similarly to the results found for RQ1, engineers will need to inspect almost the whole codebase to find all vulnerabilities. The top-10 results show some little improvements with respect to RQ1: an engineering can identify 3 to 5 vulnerable components among the top 10 suggested in `OpenSSL` and `Linux Kernel`, while 1 to 2 in `Wireshark`. However, these results are still far from those achieved in the ideal setting (RQ2). If we compare the results of Figures 5(b) and 5(c), it becomes evident that training set augmentation does not a significant difference for the system under study.

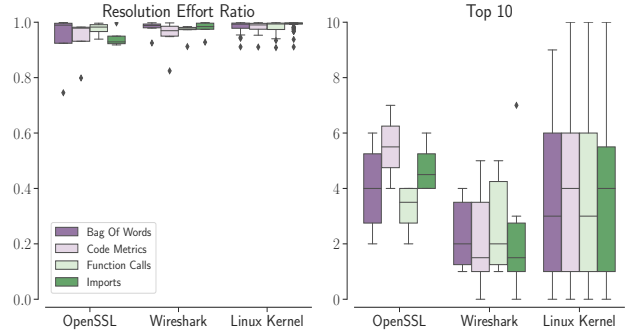
Overall these results suggest that it is possible to trade recall over precision by using training set augmentation, but overall the prediction performance do not improve much (i.e. MCC, Precision and Recall are still below 0.50 on average). A potential explanation is that bugs have a weak link with vulnerabilities (so it adds enough of noise to mislead the precision of the predictions).

8 CONCLUSION

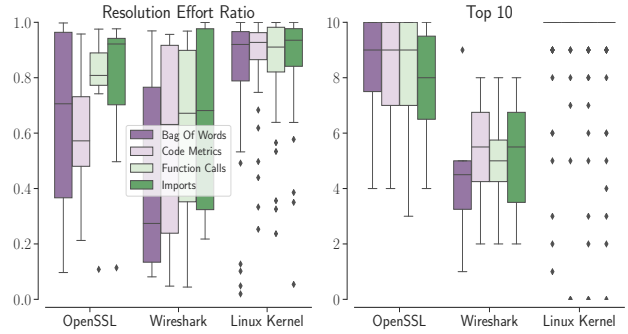
We presented a study on vulnerability prediction and showed that it has indeed good performance when trained on sufficient and accurately labelled data. However, performance is poor when considering realistic partial and mislabeled data. In particular, we found that the unrealistic use of unreported vulnerabilities at the training time gives optimistic prediction performance (MCC of 0.77, 0.65 and 0.43 for `Linux`, `OpenSSL` and `Wiresark`), which significantly degrades when using a more realistic training scenario (MCC of 0.08, 0.22, 0.10). We also showed that potential mitigation strategies, such as augmenting training data with defect information, offer little improvements. These results show that the community needs to upgrade experimental and empirical methodology for vulnerability prediction validation, evaluation and development to ensure robust and actionable scientific findings.

ACKNOWLEDGEMENTS

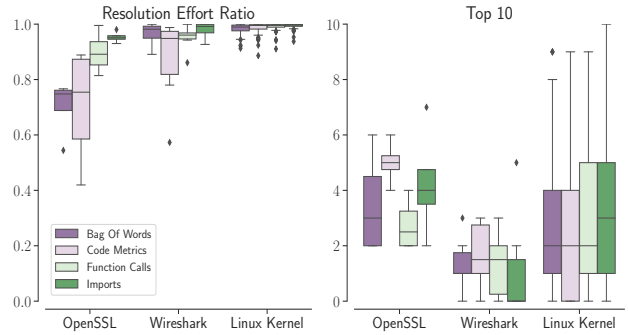
Renaud Rwemalika and Mike Papadakis are supported by the Luxembourg National Research Fund (FNR), AFR PHD 11278802 and C17/IS/11686509/CODEMATES. Mark Harman and Federica Sarro are part supported by the ERC advanced fellowship grant 741278 (EPIC: Evolutionary Program Improvement Collaborators).



(a) 'Real" world setting (RQ1). We observe a relatively low performance for all approaches and subjects. Also, all approaches show little performance differences.



(b) "Ideal" world setting (RQ2). We observe a relatively good performance for all methods and a notable performance difference with respect to the "real" world results shown in Figure 5(a).



(c) Training set augmentation in "real" world setting (RQ3). We observe that training set augmentation does not improve much the predictive performance in real world setting. It has a positive effect on `OpenSSL` (only to resolution effort ratio metric) but no effect on `Wireshark` and `Linux Kernel`. Overall training set augmentation does not lead to significant performance improvements with respect to the non-augmented results shown in Figure 5(a).

Figure 5: Resolution effort ratio and top-10 metrics for "real" world setting (a), "ideal" world setting (b) and training set augmentation in "real" world setting (c).

REFERENCES

[1] [n. d.]. Bug in OpenSSL opens two-thirds of the Web to eavesdropping. ([n. d.]). http://arstechnica.com/security/2014/04/critical-crypto-bug-in-openssl-opens-two-thirds-of-the-web-to-eavesdropping/

[2] [n. d.]. Definition of vulnerability. ([n. d.]). https://cve.mitre.org/about/terminology.html

[3] [n. d.]. Heartbleed Home Page. ([n. d.]). http://heartbleed.com

[4] [n. d.]. NATIONAL VULNERABILITY DATABASE. ([n. d.]). https://nvd.nist.gov/

[5] [n. d.]. Weka: Data Mining Software in Java. ([n. d.]). http://www.cs.waikato.ac.nz/ml/weka/

[6] Amritanshu Agrawal and Tim Menzies. 2018. Is "better data" better than "better data miners"?: on the benefits of tuning SMOTE for defect prediction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 1050–1061. https://doi.org/10.1145/3180155.3180197

[7] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. 2016. Mutation-aware Fault Prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, 330–341. https://doi.org/10.1145/2931037.2931039

[8] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Intell. Res.* 16 (2002), 321–357. https://doi.org/10.1613/jair.953

[9] Istehad Chowdhury and Mohammad Zulkernine. [n. d.]. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?. In *SAC'10*. 1963. https://doi.org/10.1145/1774088.1774504

[10] Istehad Chowdhury and Mohammad Zulkernine. 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture* 57, 3 (2011), 294 – 313. https://doi.org/10.1016/j.sysarc.2010.06.003

[11] Pedro M. Domingos. 2012. A few useful things to know about machine learning. *Commun. ACM* 55, 10 (2012), 78–87. https://doi.org/10.1145/2347736.2347755

[12] Sarro Federica. 2019. Search-Based Predictive Modelling for Software Engineering: How Far Have We Gone?. In *Proceedings of the 11th International Symposuim on Search-Based Software Engineering, SSBSE 2019*.

[13] F. Ferrucci, M. Harman, and F. Sarro. 2014. Search-Based Software Project Management. In *Software Project Management in a Changing World*. Springer, 373–399.

[14] Anthony Finkelstein, Mark Harman, Yue Jia, Federica Sarro, and Yuanyuan Zhang. 2013. *Mining App Stores: Extracting technical, business and customer rating information for analysis and prediction*. RN 13. UCL, Research Notes.

[15] Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for software analytics: Is it really necessary? *Information & Software Technology* 76 (2016), 135–146. https://doi.org/10.1016/j.infsof.2016.04.017

[16] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Trans. Softw. Eng.* 38, 6 (2012), 1276–1304. https://doi.org/10.1109/TSE.2011.103

[17] Mark Harman. 2010. The Relationship Between Search Based Software Engineering and Predictive Modeling. In *Procs. of the 6th International Conference on Predictive Models in Software Engineering (PROMISE'10)*. Article 1, 13 pages. https://doi.org/10.1145/1868328.1868330

[18] M. Harman, S. Islam, Y. Jia, L. L. Minku, F. Sarro, and K. Srivisut. 2014. Less is More: Temporal Fault Predictive Performance over Multiple Hadoop Releases. In *Proceedings of the International Symposium on Search-Based Software Engineering (SSBSE'14)*. Springer, 240–246. https://doi.org/10.1007/978-3-319-09940-8_19

[19] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2010. Fault-prone Module Detection Using Large-scale Text Features Based on Spam Filtering. *Empirical Softw. Engg.* 15, 2 (April 2010), 147–165. https://doi.org/10.1007/s10664-009-9117-9

[20] Aram Hovsepyan, Riccardo Scandariato, and Wouter Joosen. 2016. Is Newer Always Better?: The Case of Vulnerability Prediction Models. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, September 8-9, 2016*. 26:1–26:6. https://doi.org/10.1145/2961111.2962612

[21] Matthieu Jimenez. 2018. *Evaluating Vulnerability Prediction Models*. Ph.D. Dissertation. University of Luxembourg. http://orbilu.uni.lu/handle/10993/36869

[22] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. 2016. Vulnerability Prediction Models: A Case Study on the Linux Kernel. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*. 1–10. https://doi.org/10.1109/SCAM.2016.15

[23] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. 2018. [Engineering Paper] Enabling the Continuous Analysis of Security Vulnerabilities with VulData7. In *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*. 56–61. https://doi.org/10.1109/SCAM.2018.00014

[24] Igor Kononenko. 1995. On Biases in Estimating Multi-valued Attributes. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)*. 1034–1040.

[25] Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, and Federica Sarro. 2011. A Genetic Algorithm to Configure Support Vector Machines for Predicting Fault-Prone Components. In *Product-Focused Software Process Improvement - 12th International Conference, PROFES 201. Proceedings*. 247–261. https://doi.org/10.1007/978-3-642-21843-9_20

[26] B.W. Matthews. 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure* 405, 2 (1975), 442 – 451. https://doi.org/10.1016/0005-2795(75)90109-9

[27] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. 2015. Challenges with applying vulnerability prediction models. In *HotSoS'15*. 4:1–4:9. https://doi.org/10.1145/2746194.2746198

[28] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting vulnerable software components. In *CCS'07*. 529. https://doi.org/10.1145/1315245.1315311

[29] Viet Hung Nguyen and Le Minh Sang Tran. 2010. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics (MetriSec '10)*. ACM, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/1853919.1853923

[30] Federica Sarro. 2018. Predictive Analytics for Software Testing: Keynote Paper. In *Proceedings of the 11th International Workshop on Search-Based Software Testing (SBST '18)*. ACM, New York, NY, USA, 1–1. https://doi.org/10.1145/3194718.3194730

[31] Federica Sarro, Mark Harman, Yue Jia, and Yuanuan Zhang. 2018. Customer Rating Reactions Can Be Predicted Purely Using App Features. In *Proceedings of the 26th IEEE International Requirements Engineering Conference (RE'18)*.

[32] Federica Sarro, Sergio Di Martino, Filomena Ferrucci, and Carmine Gravino. 2012. A further analysis on the use of Genetic Algorithm to configure Support Vector Machines for inter-release fault prediction. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012*. 1215–1220. https://doi.org/10.1145/2245276.2231967

[33] Federica Sarro and Alessio Petrozziello. 2018. Linear Programming As a Baseline for Software Effort Estimation. *ACM Trans. Softw. Eng. Methodol.* 27, 3, Article 12 (2018), 28 pages. https://doi.org/10.1145/3234940

[34] F. Sarro, A. Petrozziello, and M. Harman. 2016. Multi-objective Software Effort Estimation. In *Procs.of the 38th International Conference on Software Engineering (ICSE'16)*. 619–630. https://doi.org/10.1145/2884781.2884830

[35] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting Vulnerable Software Components via Text Mining. *IEEE TSE* 40, 10 (Oct. 2014), 993–1006. https://doi.org/10.1109/TSE.2014.2340398

[36] Martin Shepperd, David Bowes, and Tracy Hall. 2014. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transatctions on Software Engineering* 40, 6 (2014), 603–616.

[37] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE TSE* 37, 6 (Nov. 2011), 772–787. https://doi.org/10.1109/TSE.2010.81

[38] Yonghee Shin and Laurie Williams. 2013. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering* 18, 1 (Feb. 2013), 25–59. https://doi.org/10.1007/s10664-011-9190-8

[39] Ben Smith and Laurie Williams. [n. d.]. Using SQL Hotspots in a Prioritization Heuristic for Detecting All Types of Web Application Vulnerabilities. In *ICST'11*. https://doi.org/10.1109/icst.2011.15

[40] Jeffrey Stuckman, James Walden, and Riccardo Scandariato. 2017. The Effect of Dimensionality Reduction on Software Vulnerability Prediction Models. *IEEE Trans. Reliability* 66, 1 (2017), 17–37.

[41] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*. 321–332. https://doi.org/10.1145/2884781.2884857

[42] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. 2015. Approximating attack surfaces with stack traces. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 199–208.

[43] James Walden, Jeff Stuckman, and Riccardo Scandariato. 2014. Predicting Vulnerable Components: Software Metrics vs Text Mining. In *ISSRE'14*. 23–33. https://doi.org/10.1109/ISSRE.2014.32

[44] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968

[45] Yun Zhang, David Lo, Xin Xia, Bowen Xu, Jianling Sun, and Shanping Li. 2016. Combining Software Metrics and Text Features for Vulnerable File Prediction. *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS* 2016-January (2016), 40–49. https://doi.org/10.1109/ICECCS.2015.15

[46] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *ICST'10 (ICST '10)*. IEEE Computer Society, 421–428. https://doi.org/10.1109/ICST.2010.32