
UNIVERSITY OF SALERNO

Department of Computer Science

Master of Science in Computer Science

MASTER'S DEGREE THESIS

# The Impact of Release-based Validation on Software Vulnerability Prediction Models

SUPERVISOR

Prof. **Filomena Ferrucci**

University of Salerno

CANDIDATE

**Giulia Sellitto**

0522500703

Academic Year 2019-2020

# Abstract

Developing software is a challenging process. In each phase of the evolution of a software product, security vulnerabilities can be introduced, causing severe potential threats to the system and its users. It is critical to release software that has been checked to be secure, so techniques for vulnerability discovery and analysis have to be employed. One of the most promising seems to be vulnerability prediction based on Machine Learning, and several models have been proposed by the research community. As pointed out by Jimenez et al. [15], most of those models performance have been evaluated using k-fold cross-validation, which shuffles the dataset to use for training and testing, and therefore is not realistic with respect to the real context in which such models are supposed to operate. In fact, in a real-case scenario, one would be interested in training the model using the information relative to vulnerabilities discovered in prior releases of the software and make predictions on the new code to be released. We seize this suggestion and start investigating how vulnerability prediction models would perform in a context which is more similar to the real one. We use the Walden et al.'s PHP vulnerability dataset [36] and the two models [37] based on software metrics and text tokens features, respectively. We make a comparison between the performance obtained by the two models when applying cross-validation and the performance observed when applying a release-based validation approach, which consists in using three prior releases of the application in the training phase and a single subsequent release in the testing phase [33] [32] [29]. We find that the release-based validation approach, which is more similar to a real-world vulnerability prediction process, leads to lower performance. In particular, the precision of the model drops from around 0.9 to about 0.3, which is not reasonable [23] [22].

# Contents


**1 Introduction** **1**

**2 Background** **6**

2.1 Security Vulnerabilities . . . . . 7

2.2 Machine Learning for Vulnerability Prediction . . . . . 9

**3 Related Work** **11**

**4 Empirical Study Design** **16**

4.1 Research Goal . . . . . 17

4.2 Methodology . . . . . 18

4.2.1 Dataset . . . . . 18

4.2.2 Process . . . . . 21

4.2.3 Evaluation . . . . . 24

**5 Realization** **26**

5.1 Data Preprocessing . . . . . 27

5.1.1 Software Metrics . . . . . 27

5.1.2 Text Tokens . . . . . 31

5.2 Experiments Implementation . . . . . 33

**6 Results** **35**

**7 Conclusion** **42**

# List of Figures


1.1 K-fold Cross-validation method . . . . . 3
1.2 Meaninglessness of the Cross-validation method in a real context. . . . . 3
1.3 Release-based validation method . . . . . 4

4.1 Software metrics-based model. . . . . 22
4.2 Text tokens-based model. . . . . 22
4.3 Our release-based validation approach. . . . . 23

5.1 Structure of the `filemetrics_phpmyadmin.Rda` file. . . . . 28
5.2 Structure of a software metrics table referred to a single release. . . . . 29
5.3 Structure of the `vulmovement_phpmyadmin.Rda` file. . . . . 29
5.4 Structure of an `ARFF` file containing text tokens. . . . . 32

6.1 Performance with no data balancing . . . . . 40
6.2 Performance with undersampling . . . . . 40
6.3 Performance with oversampling . . . . . 41

# List of Tables

4.1 Distribution of vulnerabilities in the dataset . . . . . 18

6.1 Summary of the experiments executions . . . . . 36

6.2 Difference between performance of different validation methods. . . . . 41

# Listings

5.1 generate_individual_metrics_files.R . . . . . 27
5.2 append_vuln_label_to_metrics.py . . . . . 30
5.3 extract_arff_tokens_archives.py . . . . . 31
5.4 arff_tokens_to_csv.R . . . . . 33
5.5 Experiments execution script . . . . . 34

# CHAPTER 1

## Introduction

In this chapter we provide an overview on our work. First, we introduce the context in which we operate and the motivations that push our research. Then, we present our objectives and the methodology that we follow, along with a brief discussion on the obtained results. Finally, we explain the structure of this thesis.

Nowadays, our society relies on software: from business transactions to healthcare services, we depend upon software systems. The current pandemic has even increased our reliance on software, leading the whole world population to interact with software for almost every social and work activity. Nevertheless, all that glitters is not gold: the security risks associated with the presence of vulnerabilities in the software are extreme. As an example, let consider the *WannaCry* ransomware attack, which exploited a vulnerability of the Windows operating system to infect more than 230,000 computers across over 150 countries during a single day, and caused total damages in the range of hundreds of millions USD [38]. Vulnerability exploits are hence a critical threat to software systems, so it is paramount to release code that has been checked to contain no vulnerabilities.

As summarized by Ghaffarian et al. [10], several approaches for vulnerability discovery have been proposed: the most promising seem to be those based on Machine Learning prediction models. These approaches are based on the fact that models can learn from vulnerabilities that are known to be present in the software, and then predict whether an unseen portion of code is likely to contain vulnerabilities. Such prediction models could be extremely useful in a software production environment, where one would be interested in training a model using vulnerability data discovered in the previous releases of the application, and make predictions on new code to be released, in order to concentrate testing effort only on those portions of source code predicted to be probably vulnerable.

Unfortunately, as pointed out by Jimenez et al. [15], most studies investigate the performance of vulnerability prediction models by using k-fold cross-validation as the evaluation method. This technique, depicted in Figure 1.1, consists in splitting the dataset in k equal-sized folds and performing k rounds of validation. In each round, k-1 folds compose the training set and one fold serves as the test set. This arrangement ensures that each fold is used for testing once and for training k-1 times. The overall performance of the model is evaluated as the average of the performance obtained in the k rounds of validation.

As explained in Figure 1.2, while this validation method can provide initial indications on the accuracy of prediction models, in a real-case scenario vulnerability data become available following time constraints, i.e., we cannot train the models on future data to obtain

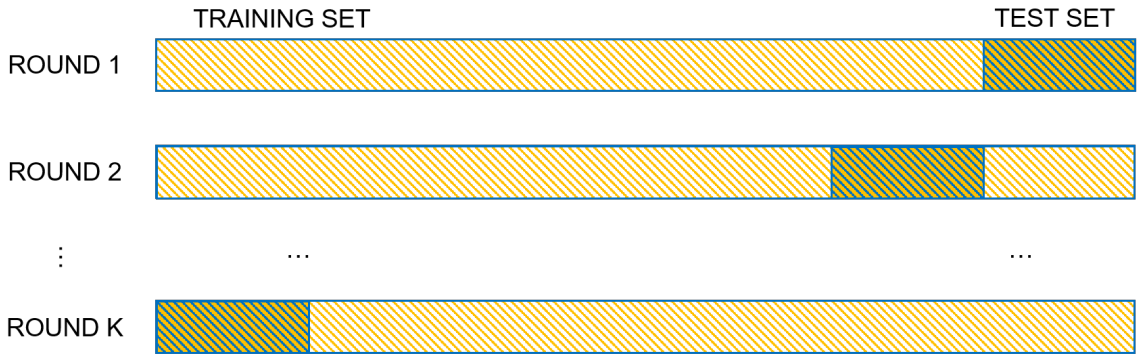



**Figure 1.1:** The k-fold Cross-validation method.

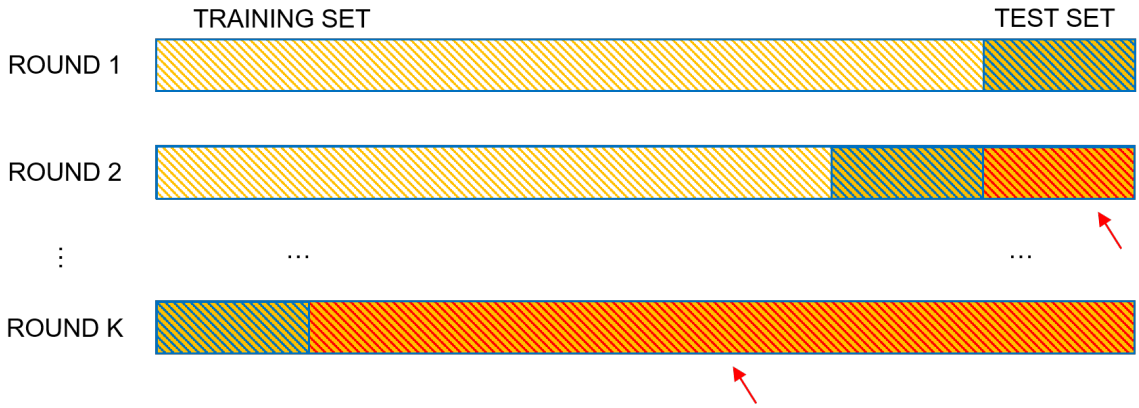



**Figure 1.2:** The meaninglessness of the Cross-validation method in a real context. In most rounds of validation, the model is trained on newer data and tested on older data.

predictions on current data, as we do by turning the folds in cross-validation. Hence, the studied effectiveness of most vulnerability prediction models is merely potential, since it is only evaluated *in vitro*. Our willing is to investigate performance of models evaluated *in vivo*, by following Jimenez et al.'s indications [15]. They proposed a more realistic approach, that consists in validating the model at a certain time $t$, using in the training phase only the information that is available at $t$, i.e., vulnerabilities already discovered and reported before $t$, just as one would do in a real-case scenario. Along with a release-based validation method, this ensures that researchers operate as closely as possible to reality.

We want to seize this suggestion and start researching in this direction, by initially focusing on investigating how software vulnerability prediction models' performance change when we use release-based validation rather than k-fold cross-validation. We perform our

study on the two models proposed by Walden et al. [37], which are based on software metrics and text tokens, respectively. These models have been evaluated by the authors on the Walden et al.'s PHP vulnerability dataset [36] using 3-fold cross-validation. We initially replicate their study, i.e., we evaluate the same models on the same dataset using cross-validation, in order to subsequently compare the performance obtained by applying a release-based validation method. Also this latter technique consists in several rounds, each focused on one release of the software. At each round, the considered release is used for the testing phase, and the model is trained using the data relative to the previous releases. The overall performance of the model is estimated considering the performance obtained in all the rounds of validation. Figure 1.3 illustrates this evaluation method.

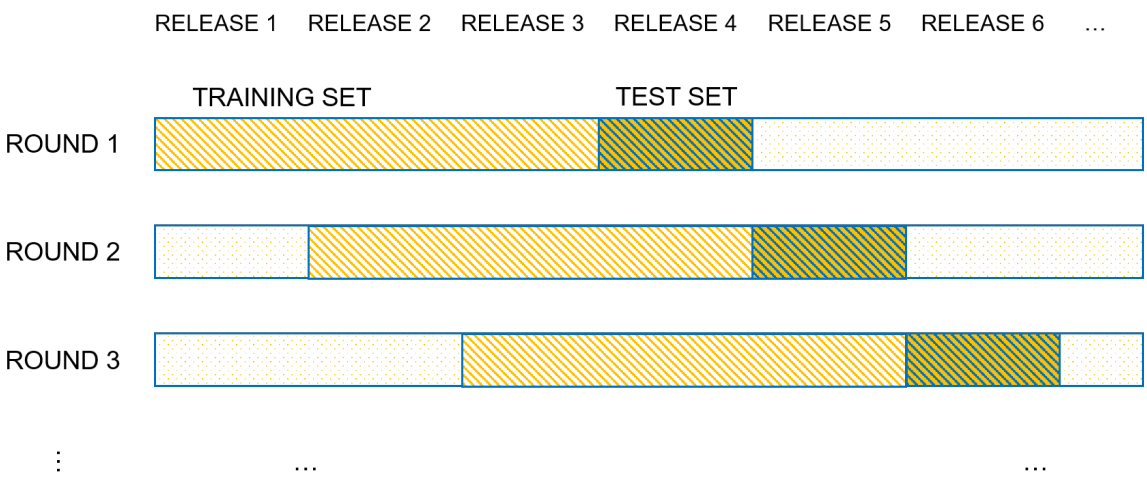



**Figure 1.3:** The release-based validation method.

We run 555 experiments to compare how the performance change when employing different models and validation methods for vulnerability prediction.

Since the dataset that we work on is highly unbalanced, i.e., the presence of vulnerable files is quite rare with respect to the total number of files, we also investigate the impact of different data balancing techniques, i.e., undersampling and oversampling. Undersampling is used in the original study by Walden et al. [37] and consists in removing from the dataset some samples of the majority class, i.e., those files that do not contain any known vulnerabilities, in order to match the number of samples in the minority class, i.e., the vulnerable files.

Oversampling consists in augmenting the number of samples in the minority class to match the size of the majority class. We follow Jimenez et al. [15] and apply the Chawla et al.'s Synthetic Minority Over-sampling TEchnique (SMOTE) [5], which adds synthesised samples to the dataset.

Our work is partly described in the extended abstract and poster that we submit to *ACM womENcourage 2021* [31]. We make the replication package publicly available [30] for other researchers to further investigate and anticipate here some of the results obtained in our study. We observe that the performance of the models evaluated with the release-based approach is lower than the performance obtained when using cross-validation. In particular, the precision achieved using the undersampled dataset drops from around 0.9 to about 0.3, and the literature suggests [23] [33] [22] that precision values over 0.7 are reasonable. Therefore, although they seem quite promising in a research context, such models are not very suitable to be employed in a real scenario.

This thesis is structured as follows: chapters 2 and 3 introduce the background and related work in the context of software vulnerability prediction models, chapters 4, 5 and 6 illustrate our study, by presenting the designed methodology, the implementation details and the observed results, and chapter 7 discusses conclusion and future goals.

# CHAPTER 2

# Background

In this chapter we introduce the main topics that our work is related to. First we draw up an overview on software security vulnerabilities and the current techniques employed to deal with them, then we discuss the new approaches based on Machine Learning.

## 2.1 Security Vulnerabilities

*"In the context of software security, vulnerabilities are specific flaws or oversights in a piece of software that allow attackers to do something malicious: expose or alter sensitive information, disrupt or destroy a system, or take control of a computer system or program."* (Dowd et al., 2007) [7]

Since the very birth of software systems, security has been a vital aspect to guarantee, and there have been several occasions in which vulnerabilities have been exploited to cause massive damage to people and businesses. One of the most famous and recent attacks is the *WannaCry* ransomware, which exploited a vulnerability of the Windows operating system to infect more than 230,000 computers across over 150 countries during a single day, and caused total loss in the range of hundreds of millions USD [38].

Vulnerability discovery is an adversary process, that puts software producers on one side and malicious attackers on the other. Software producers are interested in finding potential threats to security in their own product source code, in order to correct the weaknesses before release time. Attackers too are interested in finding flaws in the system, but to exploit them maliciously. There is an heated debate within the scientific community about vulnerability disclosures. On the one hand, publishing information on security flaws allows interested parties to patch their systems and protect themselves from possible exploits. On the other hand, also attackers get informed and can immediately take advantage of vulnerable systems in which the defect has not been corrected yet.

Nowadays, several public vulnerability registers are regularly updated. *CVE* is the Common Vulnerabilities and Exposures. As the acronym states, it is a list of publicly disclosed vulnerabilities and exposures that is maintained by the *MITRE Corporation*, a not-for-profit organization that manages federally funded research and development centers supporting several U.S. government agencies. *NVD* is the National Vulnerability Database and is maintained by *NIST*, the National Institute of Standards and Technology. Both CVE and NVD are sponsored by the U.S. Federal Government and are available for free use by anyone.

Finding vulnerabilities or potential threats in the source code of an application is not a trivial task. The problem of deciding whether a given program contains security issues or not

is called *vulnerability analysis* and it has been demonstrated to be an undecidable problem by Landi [17] and Reps [26]. Thus, one can only propose a partial solution to such problem.

At the state of the art, different techniques are employed to try to find potential vulnerabilities in the source code of an application [10]:

- Static Analysis. This approach consists in analyzing the source code without executing it. It is convenient because there are automatic tools that can be run on lots of software, and can also be integrated in the IDEs to provide immediate feedback to the developers that might be introducing security defects at implementation time [8]. These tools are good at finding a certain subset of vulnerabilities, such as buffer overflows, SQL Injection flaws, and so forth. However, many types of security problems are difficult to find automatically, for example those related to authentication. Another drawback of static analysis is the fact that often the reported vulnerabilities are actually non-vulnerable code portions.

- Dynamic Analysis. In this approach, the program is executed in order to analyze its behavior. A set of input test-cases are used to perform the analysis, and since the possible inputs are infinite, this method is not guaranteed to identify all the vulnerabilities in the code. In fact, it could be the case that a vulnerability can only be exploited with a specific input, which is not included in the test-cases set. The advantage of using such an approach is that all the reported vulnerabilities are indeed unsafe portions of code, which have been proved to be exploitable.

- Hybrid Approaches. They consist in a mixture of static and dynamic analysis techniques. For example, an approach could use static analysis to guide the selection of the test cases for dynamic analysis, hence reducing the number of tests to run. Another approach could be based on performing dynamic analysis only on those code portions that the static analysis tool pointed out as vulnerable.

## 2.2 Machine Learning for Vulnerability Prediction

Recently a further approach to vulnerability discovery and analysis has been deeply investigated. We are talking about Machine Learning algorithms, which are defined as *techniques that enable computer systems to gain new abilities without being explicitly programmed* [27]. These algorithms have been employed for highly difficult problems, which algorithmic solutions are impractical. The concept of gaining new abilities has been illustrated by Mitchell, who formulated the following definition [21]:
*"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, measured by P, improves with experience E."*

In other words, machine learning techniques make it possible for computers to execute some tasks without strictly following a sequence of instructions, as traditional algorithms do. What they actually do is building a statistic model of the data they receive as input, in order to *learn* their intrinsic characteristics and be able to execute a specific task. In the field of vulnerability analysis, we talk about the task of *vulnerability prediction* when applying the use of machine learning.

Vulnerability prediction is typically a classification task, in which we want to label certain portions of the application source code as *vulnerable* if they contain vulnerabilities, or as *neutral* if they do not show the presence of any known security defects. The labelling can be performed at different granularity levels:

- File level. Each source code file is labelled as *vulnerable* or *neutral*.
- Class level. Relative to object oriented contexts, in which one can focus their attention on single classes.
- Function or Method level. The finest granularity, at which the labelling is relative to individual functions or methods.

The classification task is usually performed by employing supervised learning approaches. In supervised learning, we provide to the machine several samples of data which we already labelled. The machine is able to build a model of such data and to learn the characteristics

useful to discern between *vulnerable* and *neutral* elements. This is called the training phase, and the output is the model which can be queried to make predictions. We can then submit a new element, of which we do not have any information about the presence of vulnerabilities, and the model can predict whether the element is *vulnerable* or *neutral*.

The advantage of vulnerability prediction is that developers can focus their attention on those elements predicted as *vulnerable* and deepen the testing effort, in order to detect the security flaws and correct them before releasing the software.

# CHAPTER 3

## Related Work

In this chapter we provide an overview of the state of the art in the topic of software vulnerability prediction using Machine Learning models.

Software vulnerability prediction using Machine Learning has been an *hot topic* for many years [10]. Several prediction models have been proposed by the research community, whose members are interested in analyzing how effectively they would perform in a software developing context. One of the first studies investigating a Machine Learning approach for vulnerability prediction is by Neuhaus et al. [23]. They focus their work on the Mozilla Firefox project and try to consider file imports, function calls and past information as predictors of vulnerability at component level. They note that components and functions previously reported as containing a single vulnerability in the past are not likely to contain further vulnerabilities in the future, meaning that the single fix lasts over time. They find out that components having similar file imports and function calls are likely to be vulnerable. Vulnerability prediction at the component granularity level is recently investigated also by Sultana et al. [34], who focus on classes and methods in Java projects.

However, most of the vulnerability prediction models that have been proposed during the years operate at file-level granularity [10]. Meneely and Williams [19] [20] consider 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 investigate the Linus' Law defined by Eric Raymond [25] as *"Given enough eyeballs, all bugs are shallow"*, which means that if many people work on a software project, it is likely to be secure. Opposite to this Law there is the *"too many cooks in the kitchen"* principle, which points out that a high number of people working on the same project can increase the possibility of introducing defects. Meenely and Williams find out that the developer activity metrics can be used as indicators for source code file vulnerability. They also work with Shin [33] [32] to deeper explore the predictive power of other metrics, such as complexity metrics and code churn metrics. They base their investigation on the idea that complex code is difficult to understand and test, and frequent or large code changes can introduce vulnerabilities.

Complexity metrics are also studied by Walden et al. [37], along with other product metrics, such as number of lines of code, indicators of code volume and so on. They propose two models for vulnerability prediction, one based on software metrics and the other based on text tokens. The latter uses the *Bag of Words* approach, which the authors also apply in

previous work [11] [28] and which consists in the following steps. First, the vocabulary of the application is built, by storing all the text tokens that appear in the source code files, i.e., keywords of the language, literals and other code constructs. Then, each file is represented by the frequency vector, which stores the number of occurrences of each word of the vocabulary in the considered file. The frequency vectors are used as predictors for the model. Walden et al. compare the performance obtained from the two models evaluated with 3-fold cross-validation on a dataset that they build. The dataset [36] contains vulnerabilities found in several releases of three open-source PHP projects, i.e., PHPMyAdmin, Moodle and Drupal.

Walden et al.'s study and dataset [37] constitute a milestone in the research context of software vulnerability prediction.

Chen et al. [6] carry out a large-scale empirical analysis using the same PHP vulnerability dataset to perform their experiments. They consider 48 different prediction models, including also the ones based on software metrics and text tokens, i.e., Bag of Words, and aim at determining the best method in terms of output performance. Kaya et al. [16] operate on the same dataset to evaluate the impact of choosing different settings when building a model. They compare the performance obtained considering 7 classifiers and 4 data balancing techniques to deal with the imbalance of the dataset. They too employ Walden et al.'s models, using software metrics, text tokens and a combination of the two as predictors. Also Zhang et al. [40] experiment the utilisation of the two predictors jointly, proposing an original approach, called *VULPREDICTOR*. It is a composite prediction model that operates on two levels. First it builds 6 underlying independent classifiers, and then it constructs a meta-classifier to combine the outputs of the 6 underlying classifiers. Another multi-level solution is proposed by Catal et al. [3], who deploy a vulnerability prediction web service on Microsoft Azure cloud computing platform. The service takes software metrics as predictors and, after performing some steps of data cleaning an preparation, it feeds data to a stratified neural network for vulnerability prediction.

In all the aforementioned researches, the proposed models' performance is evaluated on the Walden et al.'s dataset [36] by using k-fold cross-validation. This method consists in splitting the available dataset in k slices, i.e., folds, and perform k rounds of validation. In

each round, k-1 folds constitute the training set, that the model exercises on in the learning phase, and the remaining fold makes up the test set, that the model is evaluated onto. The overall performance of the model is computed as the average of the performance observed in each round. This approach is widely used in the field of Machine Learning, because the dataset splitting and rotation avoids performance measure bias due to possible lucky conformations of the training and test data.

However, in a recent study published in 2019, Jimenez et al. [15] highlight a problem that arises from the application of cross-validation to evaluate vulnerability prediction models. They argue 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 to train 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. compare 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 discover 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.

Other studies investigate release-based validation approaches, but still under the *perfect labelling assumption*. Scandariato et al. [29] focus their work on 20 Android applications and employ the Bag of Words method based on text tokens. Also Jimenez et al., in a previous study [14] on the Linux kernel dataset, apply a release-based validation method but without considering the mislabelling noise.

Vulnerability prediction is a crucial topic and the research community is currently committed to it. Recent works [4] also suggest the use of Deep Learning models to enhance the predictive power of the existing proposals. Additional solutions to the vulnerability

prediction problem consist in original approaches, for example the one proposed by Garg et al. [9] in 2020. They develop a novel method based on Machine Translation: an automatic translator is trained with vulnerable samples of code and their respective fixes, and is capable of producing a corrected version of new samples which are given as input to it. The prediction consists in feeding the translator unseen source code, and if it generates a translation, i.e., a fix, the code is likely to be vulnerable.

# CHAPTER 4

## Empirical Study Design

In this chapter we present our empirical study. First, we introduce the research goal that we keep in mind during our investigation. Then, we discuss the methodology that we use to perform our study.

## 4.1 Research Goal

Our study consists in taking some preliminary steps to get from the traditional cross-validation approach, which has been broadly used to evaluate vulnerability prediction models in research contexts, towards the realistic one suggested by Jimenez et al. [15]. In a real-case scenario, one would train the model using only the data referred to vulnerabilities already discovered in prior releases of the software and make predictions on the latest code to be released. Our goal is to investigate how a release-based validation method would impact the performance of models formerly evaluated using cross-validation. We focus our research on the two models proposed by Walden et al. [37], i.e., the one based on software metrics and the one based on textual tokens. These models have been evaluated by the authors using 3-fold cross-validation on a dataset composed of three popular open-source PHP applications, i.e., PHPMyAdmin, Moodle and Drupal [36]. We want to use the same models and dataset to initially replicate their study and then analyze the difference in the performance evaluated with a release-based approach. Thus, we formulate the following research questions:

**RQ$_1$**: What is the performance of vulnerability prediction models validated using a release-based approach when compared to models evaluated using cross-validation?

**RQ$_2$**: Which modelling approach is more sensitive to the use of a different validation method?

Since the dataset is highly unbalanced, i.e., vulnerabilities are not much frequent, we want to examine the use of data balancing techniques. Walden et al. [37] use undersampling, that is, removing samples from the majority class in order to equalize the number of elements in the minority class. Jimenez et al. [15] also investigate the use of oversampling, i.e., augmenting the minority class by adding to the dataset several synthetic vulnerable elements. We want to study the impact of such techniques on the models' performance, hence we make the following research question:

**RQ$_3$**: Which data balancing technique leads to better performance?

| Vulnerability Category | Number of Samples |
|---|---|
| Code Injection | 10 |
| CSRF | 1 |
| XSS | 45 |
| Path Disclosure | 12 |
| Authorization Issues | 6 |
| Other | 1 |
| Total | 75 |

**Table 4.1:** Distribution of vulnerabilities in the dataset

## 4.2 Methodology

### 4.2.1 Dataset

For our research we use the Walden et al.'s PHP vulnerability dataset [36] which contains vulnerability data mined from three popular open-source PHP applications, i.e., PHPMyAdmin, Moodle and Drupal. The dataset was published in 2014 along with Walden et al.'s proposal of two vulnerability prediction models [37], one based on software metrics and the other based on text tokens. Since then, it has been used in over 90 researches [2] and in every single one of those works cross-validation is applied to evaluate the proposed models.

We perform our work on the PHPMyAdmin dataset. PHPMyAdmin is one of the most popular database administration tools [1] and allows managing MySQL databases over the web. The dataset includes data relative to 95 releases of the application, between 2.2.0 and 4.0.9. The original authors collected source code files of major (e.g. v2.0), minor (v2.1), and subminor (v2.1.5) versions of the application from the public Git repository and mined vulnerability information from the security announcements maintained by the project itself.

The dataset contains 75 vulnerabilities, categorized by the authors as follows [37]:

- 10 samples of Code Injection. These vulnerabilities allow attackers to modify server-side variables or HTTP headers, or execute code on the server.

- 1 sample of CSRF. Cross-site Request Forgery induces the authorized user to perform unintended actions wanted by the attacker.
- 45 samples of XSS. Cross-site Scripting allows malicious Javascript code to be executed in the browser of the user.
- 12 samples of Path Disclosure. These vulnerabilities allow malicious exploiters to obtain the installation path of the application. This information can be useful to perform a subsequent attack.
- 6 samples of Authorization Issues. The vulnerabilities included in this category are general violations of the CIA triad, i.e., confidentiality, integrity, or availability, not falling in other categories. For example, vulnerabilities related to access control and information disclosure in general.
- 1 sample of a no better-specified vulnerability.

The dataset provides a tracking matrix that keeps record of which files were affected by each vulnerability, at the time of each considered release. Some of the vulnerabilities migrate among versions, since time passes from their introduction in the code to their discovery and fix. Table 4.1 offers an overview of the dataset and the distribution of vulnerabilities in the application.

The authors used the dataset to experiment their proposed vulnerability prediction models, i.e., the one based on software metrics and the other based on text tokens. They provide the data used for their experiments, for other researchers to work with.

For the vulnerability prediction model based on text tokens, the authors used the PHP built-in `token_get_all` function to collect tokens from each source code file. Tokens represent language keywords, punctuation, and other code constructs. Comments and whitespaces are ignored, and literals are converted into fixed tokens, e.g., `T_STRING` is used to represent a string literal instead of the contents of the string. The tokens string representing each file of each release is available in the dataset.

For each file of each version, the following software metrics were computed with a tool specifically developed by the authors:

- 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). This is the same as Lines of code, except HTML content embedded in PHP files, i.e., content outside of PHP start/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. This metric is computed by adding one to the number of loop and decision statements in the file.
- Maximum Nesting Complexity: The maximum depth to which loops and control structures in the file are nested.
- Halstead's Volume. The volume estimated as the file's vocabulary multiplied by the logarithm of the file length. The vocabulary is given by the sum of the number of unique operators and the number of unique operands. Operators are method names and PHP language operators, operands are parameter and variable names. The file lenght is given by the sum of the total number of operators and the total number of 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.

The metrics computed for each file of each release are available in the dataset.

### 4.2.2 Process

Our work consists in analyzing the difference in the performance of the vulnerability prediction models observed when using a release-based validation approach rather than the cross-validation method.

We study Walden et al.'s [37] models, i.e., the one based on software metrics and the one based on text tokens. These models perform file-level vulnerability prediction: 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. We have to point out that files are never labelled as *non-vulnerable*, since we cannot be sure that they do not contain any vulnerabilities at all. In fact, they could contain vulnerabilities that have not been discovered yet.

The model based on software metrics is depicted in Figure 4.1 and it works as follows. Each file is represented by a set of numeric values, i.e., the software metrics computed by the authors [37]. These values constitute the independent variable the classifier takes as input. The dependent variable is the label *vulnerable* or *neutral*. The classifier is able to predict the dependent variable basing on the independent variables' values.

The model based on text tokens employs the *Bag of Words* approach. Each source code file is *tokenized*, that is, converted into a list of tokens. Tokens represent language keywords, punctuation, and other code constructs, in addition to variable names and literals. The output of the tokenization process is made available by the authors [37]. All the tokens generated from all the files make up the vocabulary of the application, which is used to create the words frequency vector of each file. The frequency vector of a file stores the number of occurrences of each word of the vocabulary in the considered file. The values in the frequency vector constitute the independent variable for the classifier. Figure 4.2 depicts this model.

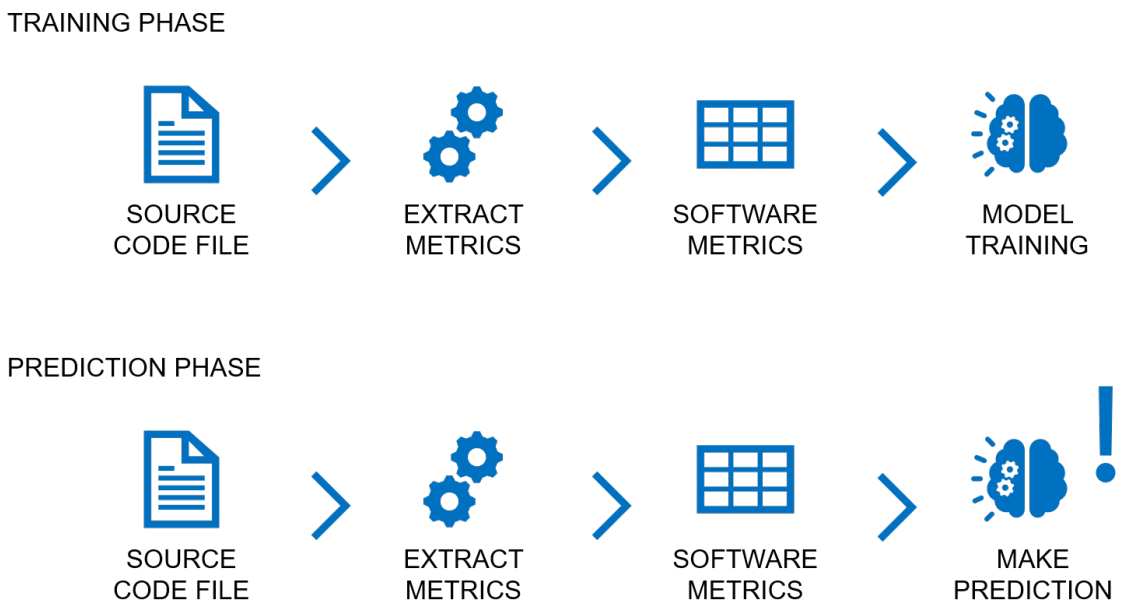



**Figure 4.1:** Software metrics-based model.

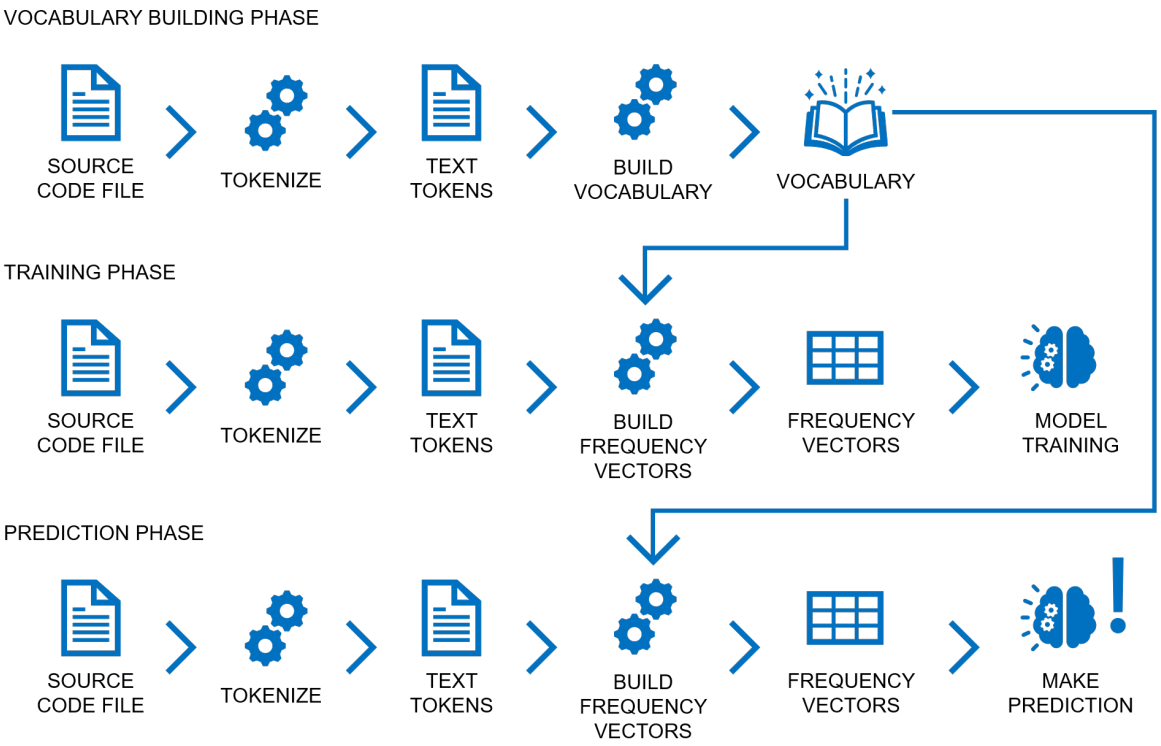



**Figure 4.2:** Text tokens-based model.

The classifier considered by the original authors [37] and in this study is Random Forest. This algorithm is the most used and better preferred [10] [6] [16], since it enhances the Decision Tree approach and corrects its problems with overfitting.

Our work consists in the following phases: first we execute a series of experiments to evaluate the performance of the two models by using 3-fold cross-validation, as done by the original authors [37]. For each model, i.e., the one based on software metrics and the one based on text tokens, the PHPMyAdmin dataset is split into 3 equally large folds and 3 rounds of validation are executed. The overall performance of each model is computed as the average performance obtained in the 3 rounds. Then, we focus on the release-based validation approach. We perform several experiments by considering one release per time in the validation set. The training set is made up of a certain number of releases which are prior to the one each experiment is focused on. We follow the methodology suggested by Shin and Williams et al. [33] [32], by including 3 prior releases in the training set. Thus, we start by using the first three releases for training and the fourth for testing, and we proceed as depicted in Figure 4.3, resulting in the execution of several experiments for each model.

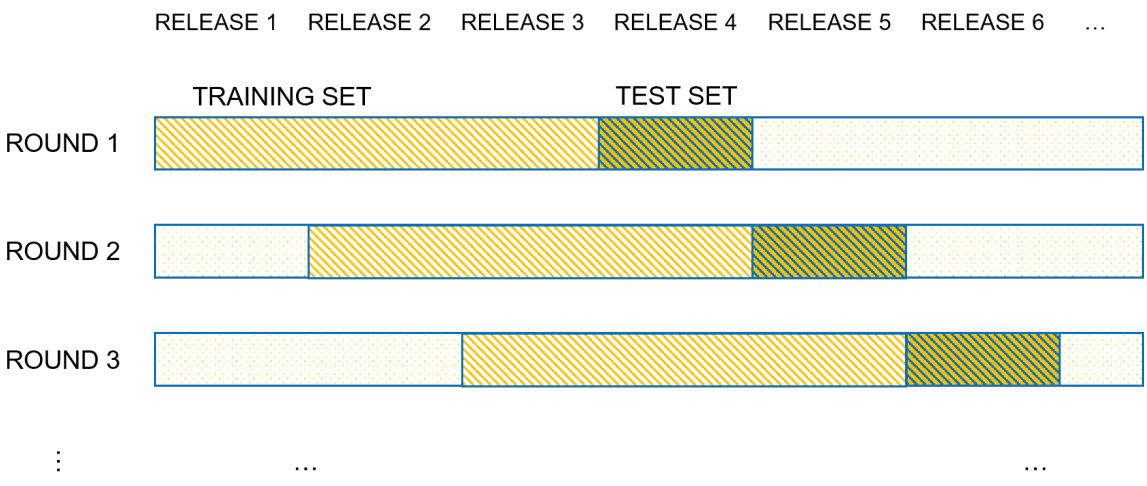



**Figure 4.3:** Our release-based validation approach.

Since the dataset is highly unbalanced and our 3rd Research Question pushes us to investigate data balancing techniques, we repeat all the experiments 3 times: once we use the dataset as-is, once we perform undersampling and once we perform oversampling. Undersampling consists in removing samples from the majority class, i.e., the neutral files

subset, in order to match the number of samples in the minority class, i.e., the vulnerable files. Oversampling consists in augmenting the number of samples in the minority class to match the size of the majority class. To perform oversampling we follow Jimenez et al. [15] and use the Chawla et al.'s Synthetic Minority Over-sampling TEchnique (SMOTE) [5], which adds synthesised samples that have similar features as the real instances of the minority class.

### 4.2.3 Evaluation

We base the performance evaluation on the confusion matrix, which summarizes the predictions provided by the model. 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 contain vulnerabilities. The sum of True Positives, True Negatives, False Positives and False Negatives gives the number of total predictions output by the model.

We use the following measures as indicators of performance:

- *Precision*. Indicates the percentage of actual vulnerable files among the ones predicted to be vulnerable and is given by the formula:

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

  An high precision means that the model is often right when it classifies a file as vulnerable. The literature suggests [15] [23] [33] [22] 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. [37]:

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

- *F1-score*. It is the geometric 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:

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

# CHAPTER 5

## Realization

In this chapter we illustrate the realization of our empirical study. First we discuss the data preparation and preprocessing phase, then we describe the implementation of the experiments.

The code that we implement for the realization of our research is available in the replication package [30].

For our work we decide to use Python language. It is highly intuitive and easily readable, and it offers several packages, i.e., libraries, which provide implementations of the most popular Machine Learning algorithms. The main data structure used in such implementations is the `pandas data frame`, which keeps data in a tabular form. This structure can be easily saved to and read from files using the `CSV` format.

## 5.1 Data Preprocessing

The dataset we work with [36] provides diverse data in various formats. We first perform some steps of data preprocessing and preparation in order to uniform the formats and facilitate the next implementation phases.

### 5.1.1 Software Metrics

Software metrics computed for all files of all releases of the considered application are available in a single `Rda` file. `Rda` stands for `R Data` format and it is designed to be used in R, which is a system for statistical computation. The `filemetrics_phpmyadmin.Rda` file provided in the dataset contains a single structure, called `filemetrics`, that stores 94 further substructures. Each one of those substructures is a table that keeps the data relative to one release of the application. In such tables, each row represents a file of the corresponding release, and columns represent software metrics values. Figures 5.1 and 5.2 show the original structure of the `filemetrics_phpmyadmin.Rda` file that we just described.

In order to properly work with such data in Python, we first need to perform some steps of data preprocessing and preparation. The `filemetrics` structure and all the sub-structures relative to releases are not suitable to be used in Python, so we decide to store each table in a separate `Rda` file, resulting in 94 individual files, each of them containing a software metrics table. We use the R script in Listing 5.1 to perform this task. At line 4 we load the complete `filemetrics` structure, from which we retrieve information of each single release at line 9.

| Name | Type | Value |
|---|---|---|
| filemetrics | list [94] | List of length 94 |
| RELEASE_2_2_1 | double [68 x 13] | 134.0 15.0 276.0 107.0 163.0 4.0 134.0 15.0 331.0 134.0 168.0 ... |
| RELEASE_2_2_2 | double [70 x 13] | 141.0 15.0 300.0 112.0 163.0 8.0 141.0 15.0 357.0 139.0 168.0 1 ... |
| RELEASE_2_2_3 | double [74 x 13] | 144.0 15.0 321.0 112.0 163.0 8.0 144.0 15.0 380.0 139.0 168.0 1 ... |
| RELEASE_2_2_4 | double [79 x 13] | 152.0 15.0 341.0 112.0 163.0 8.0 152.0 15.0 408.0 139.0 168.0 ... |
| RELEASE_2_2_5 | double [82 x 13] | 153.0 15.0 341.0 112.0 164.0 8.0 153.0 15.0 408.0 139.0 169.0 ... |
| RELEASE_2_2_6 | double [87 x 13] | 157.0 15.0 341.0 112.0 165.0 8.0 157.0 15.0 412.0 139.0 170.0 ... |
| RELEASE_2_3_0 | double [167 x 13] | 7.0 260.0 16.0 29.0 38.0 64.0 8.0 260.0 16.0 32.0 38.0 85.0 0.0 ... |
| RELEASE_2_3_1 | double [170 x 13] | 7.0 268.0 16.0 32.0 38.0 75.0 8.0 268.0 16.0 35.0 38.0 10 ... |
| RELEASE_2_3_2 | double [171 x 13] | 7.0 268.0 16.0 32.0 38.0 75.0 8.0 268.0 16.0 35.0 38.0 10 ... |
| RELEASE_2_3_3 | double [173 x 13] | 7.0 268.0 14.0 142.0 32.0 38.0 8.0 268.0 14.0 157.0 35.0 3 ... |

**Figure 5.1:** Structure of the `filemetrics_phpmyadmin.Rda` file, as visualized in the R environment. The main data structure is called `filemetrics` and contains a list of tables, each referring to a PHPMyAdmin application release.

```
# [...input and output files path retrieval...]

# load metrics structure
load(file = inputfile)

# save metrics table of each release in a separate file
for (releasename in names(filemetrics)) {
    singlefilename = paste(output_dir,"/",releasename,".Rda", sep = "")
    metrics = filemetrics[[releasename]]
    save(metrics, file = singlefilename)
}
```

**Listing 5.1:** `generate_individual_metrics_files.R`

To perform our main task and give the software metrics data to the vulnerability prediction model, we need to build the complete feature matrix. This consists in creating a table that stores both the independent variables, i.e., the software metrics, and the dependent variable, i.e., the *vulnerable/neutral* label, for each source code file of the dataset. We retrieve information about the vulnerability label of each file in the dataset by using the vulnerability tracking data stored in the `vulmovement_phpmyadmin.Rda` file, whose table structure is

shown in Figure 5.3. For each vulnerability tracked in the dataset, a row appears in the table. The identifier of the row is the hash of the commit that introduced the vulnerability. Each column is referred to a release of the application and holds the name of the file in the release that is affected by the vulnerability. For example, we can see in the fragment in Figure 5.3 that the first tracked vulnerability affects the `libraries/Error.class.php` file in releases 3.1.0, 3.1.1 and 3.1.2. The last shown vulnerability is not present in these three releases, so it is either already corrected before release 3.1.0 or introduced after release 3.1.2.

| | nonecholoc | loc | nmethods | ccomdeep | ccom | nest | hvol |
|---|---|---|---|---|---|---|---|
| **config.inc.php3** | 141 | 141 | 0 | 1 | 1 | 0 | 325.43452 |
| **db_create.php3** | 15 | 15 | 0 | 3 | 3 | 1 | 124.82542 |
| **db_details.php3** | 300 | 357 | 0 | 57 | 57 | 7 | 6067.92950 |
| **db_printview.php3** | 112 | 139 | 0 | 33 | 33 | 7 | 1640.64548 |
| **db_stats.php3** | 163 | 168 | 1 | 24 | 24 | 5 | 2773.76037 |
| **footer.inc.php3** | 8 | 10 | 0 | 4 | 4 | 1 | 54.93061 |
| **header.inc.php3** | 67 | 80 | 0 | 12 | 12 | 3 | 1010.76133 |
| **index.php3** | 22 | 23 | 0 | 5 | 5 | 4 | 238.29756 |
| **lang/arabic.inc.php3** | 307 | 307 | 0 | 1 | 1 | 0 | 1751.41704 |
| **lang/brazilian_portuguese.inc.php3** | 307 | 307 | 0 | 1 | 1 | 0 | 1751.41704 |

**Figure 5.2:** Structure of a software metrics table referred to a single release. Each row represents a file in the release, columns store the metrics values.

| | 3_1_0 | 3_1_1 | 3_1_2 |
|---|---|---|---|
| **50cd1e930fd048348b34da948f1309a1d951706f** | libraries/Error.class.php | libraries/Error.class.php | libraries/Error.class.php |
| **e623dc42cf635cfd8d3d6644b04144cfe02e588c** | libraries/Error_Handler.class.php | libraries/Error_Handler.class.php | libraries/Error_Handler.class.php |
| **d0facc3d4fe3a7594e38163cc75fd2da7c734fa7** | libraries/common.inc.php | libraries/common.inc.php | libraries/common.inc.php |
| **19a5d15d30bebaf3e39602babc33c149078d9cbf** | pmd_common.php | pmd_common.php | pmd_common.php |
| **b098a846394bc8a925567c9dc35e8a0ed9b35683** | pdf_schema.php | pdf_schema.php | pdf_schema.php |
| **f932ee45ed1d6c2d057071db7afbc2f6d7201608** | | | |
| **094702ba10ce79ed5fa403de55b7dccdb63e9165** | | | |
| **3f943b47048f1d48997354cd4e7b8c49bed76bf2** | view_create.php | view_create.php | view_create.php |
| **7b48016c66a422fe39c509728840327fb11f6502** | show_config_errors.php | show_config_errors.php | show_config_errors.php |
| **9bd9a44a6795f3c7c0972f06cac351ad051240bf** | | | |

**Figure 5.3:** Structure of the `vulmovement_phpmyadmin.Rda` file, as visualized in the R environment. For each vulnerability tracked in the dataset, a row appears in this table. The identifier of the row represents the hash of the commit that introduced the vulnerability, and the columns store information about the affected files.

```
# [...input and output files path retrieval...]

# for each release in the vulnerability tracking table:
for release in vulmovement.columns:

    # store vulnerable files names
    vulnerable_files = []
    for filename in vulmovement[release]:
        if filename:
            vulnerable_files.append(filename)

    try:
        # extract the file metrics dataframe from file
        obj = pyreadr.read_r(os.path.join(r_metrics_dir, "RELEASE_" +
    release + ".Rda"))
        metrics_df = obj["metrics"]

        # build column to store each file's vulnerability label
        # first build an all-NEUTRAL column
        metrics_df["IsVulnerable"] = ["no"] * len(metrics_df.index)

        # add VULNERABLE label to rows corresponding to vulnerable files
        metrics_df.loc[vulnerable_files, ["IsVulnerable"]] = "yes"

        # save metrics with labels in a new file
        metrics_df.to_csv(os.path.join(output_dir, "RELEASE_" + release +
     ".csv"))

    except pyreadr.custom_errors.PyreadrError:
        # not all releases metrics are stored
        continue
```

**Listing 5.2:** `append_vuln_label_to_metrics.py`

Listing 5.2 shows the Python script we use to build the complete feature matrix for each release of the application. We start from the `vulmovement` structure and proceed by columns, i.e., by releases. We first retrieve the names of all the files that are affected by some vulnerabilities in the considered release. Then we add the `IsVulnerable` column to the software metrics table of the release, and populate it by setting `yes` in correspondence of the vulnerable file names, and `no` elsewhere. The resulting feature matrix is saved into a `CSV` file and is ready to be easily used in Python.

### 5.1.2 Text Tokens

Text tokens are provided in the dataset as individual `ARFF` files embedded in `GZ` archives. The `GZ` format is the `GNU Zipped Archive` format, used in most `Unix-like` systems. The `ARFF` acronym stands for `Attribute Relationship File Format` and it is used to store databases in `Weka`, which is a software for data analysis and predictive modeling.

To work with text tokens data in Python, we first extract all the individual files from the archives, and then convert the `ARFF` files into `CSV`, to guarantee that all the data we use is represented uniformly. Listing 5.3 shows the Python script we use for archives extraction.

```
# [...input and output files path retrieval...]
# input_dir contains all the archives to extract

for filename in os.listdir(input_dir):
    archive_filename = os.path.join(input_dir, filename)
    release_str = filename.split("-")[1]
    output_filename = os.path.join(output_dir, "RELEASE_" + release_str +
    ".arff")

    with gzip.open(archive_filename, 'rb') as f_in:
        with open(output_filename, 'wb') as f_out:
            # extraction to the target
            shutil.copyfileobj(f_in, f_out)
```

**Listing 5.3:** `extract_arff_tokens_archives.py`

The extraction results in a series of individual ARFF files, each of them referring to a single release of the application. Figure 5.4 shows a portion of the content of one of such ARFF files. The preamble of the file explains the database structure. Each row is referred to a source code file in the release of the application. The first column stores all the text tokens of the file, and the second column contains the vulnerability label, i.e., keeps the IsVulnerable attribute.

```
@relation R_data_frame

@attribute tokens string
@attribute IsVulnerable {no,yes}

@data
'T_OPEN_TAG T_VARIABLE($cfgPmaAbsoluteUri) = ; T_VARIABLE($cfgServers) [
T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE
($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ]
= ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [
T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = T_STRING ;
T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [
T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE
($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ]
= ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [
T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE
($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ]
= ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [
T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = T_STRING ;
T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [
T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE
($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ]
= ; T_VARIABLE($cfgServers) [ T_LNUMBER ] [ ] = ; T_VARIABLE($cfgServers) [
```

**Figure 5.4:** Structure of an ARFF file containing text tokens of all the files in a single release. Each row of the database represents a source code file of the release. The first column stores all the text tokens, the second column contains the vulnerability label.

Since we cannot work with ARFF files in Python, we use the R script in Listing 5.4 to convert such files in the CSV format. The foreign library is needed to interface with different file formats. The first step of the conversion is performed at line 15 and consists in reading the ARFF file content and storing it in an R data frame, i.e., table. This table can be built by referring to the database structure defined in the preamble of the ARFF file. At line 18 the conversion is finalized: the table is saved to a CSV file.

All the data is now ready to be used for our experiments in Python.

```r
# to work with different file formats
library(foreign)

# [...input and output files path retrieval...]
# inputdir is the directory containing all the .ARFF files

previouswd = getwd()
setwd(inputdir)

# retrieve all the .ARFF files names
allinputfilenames = list.files()

for(i in 1:length(allinputfilenames)) {
    # convert .ARFF to .CSV
    mydata=read.arff(allinputfilenames[i])
    filename = strsplit(allinputfilenames[i], ".arff")
    outputfile = paste(outputdir,"/",filename,".csv", sep = "")
    write.csv(mydata, outputfile)
}

# revert to the previous working directory
setwd(previouswd)
```

**Listing 5.4:** `arff_tokens_to_csv.R`

## 5.2 Experiments Implementation

To allow the batch execution of all the experiments that we planned in the empirical study design, we implement the code shown in Listing 5.5.

The `generate_all_experiments_settings` method at line 5 plans the execution of each experiment, by specifying:

- the dataset to work with, i.e., PHPMyAdmin;

- the approach to employ, i.e., software metrics or text tokens;
- the validation method to use, i.e., 3-fold cross-validation or release-based;
- the releases to consider in the training and test set, respectively;
- the data balancing technique to apply, i.e., undersampling, oversampling or none;
- the classifier to use, i.e., Random Forest.

The performance indicators output from each experiment execution are kept by a `Log` object, which also stores information about the amount of time the model took for training. All the logs are saved to a `CSV` file, in order to allow querying and analyzing the results of the experiments.

```
# prepare csv file for experiments output
pandas.DataFrame(Log.header()).to_csv(output_file, index=False)

# generate all experiments to run
experiments_to_run = generate_all_experiments_settings()

for i in range(0, len(experiments_to_run)):
    # launch the experiment execution
    log = execute_experiment(i+1, experiments_to_run[i])
    # save results: append log to the csv file
    pandas.DataFrame(log).to_csv(output_file, mode='a', index=False,
    header=False)

print("All done!")
print(str(len(experiments_to_run)) + " experiments saved to file: " +
    output_file)
```

**Listing 5.5:** Experiments execution script

# CHAPTER 6

# Results

In this chapter we illustrate the results of the experiments and discuss our answers to the research questions that we formulated in chapter 4.

We execute a total of 555 experiments in order to collect information about the performance of the considered software vulnerability prediction models and provide answers to our research questions. Table 6.1 summarizes the experiments. We point out that 95 releases of the PHPMyAdmin application are available in the dataset [36], but the data relative to software metrics of release 2.0.0 is missing. When applying the release-based validation method, we start testing from the fourth release onwards, so the first three rounds of validation are omitted. These are the reasons why we perform 91 and 92 release-based validation experiments with the software metrics and text tokens approaches, respectively.

| Validation Method | Modelling Approach | Data Balancing Technique | Number of Experiments |
|---|---|---|---|
| Cross-validation | Software Metrics | none | 1 |
| Cross-validation | Software Metrics | undersampling | 1 |
| Cross-validation | Software Metrics | oversampling | 1 |
| Cross-validation | Text Tokens | none | 1 |
| Cross-validation | Text Tokens | undersampling | 1 |
| Cross-validation | Text Tokens | oversampling | 1 |
| Release-based | Software Metrics | none | 91 |
| Release-based | Software Metrics | undersampling | 91 |
| Release-based | Software Metrics | oversampling | 91 |
| Release-based | Text Tokens | none | 92 |
| Release-based | Text Tokens | undersampling | 92 |
| Release-based | Text Tokens | oversampling | 92 |
| | | Total | 555 |

**Table 6.1:** Summary of the experiments executions.

The performance obtained from the execution of the experiments are depicted in Figures 6.1, 6.2 and 6.3. In order to make the comparison straightforward, we plot in the same figure the performance output by the same model when evaluated with the two different methods. The dots indicate the performance obtained in 3-fold cross-validation, the box plots summarize the performance collected in all the rounds of release-based validation. The plots referring to the two models, i.e., the one based on software metrics and the one based on text tokens, are put side to side. The three sets of plots group together the performance observed when using different data balancing techniques.

By analyzing the performance plots, we can discuss the answers to our research questions.

**RQ**$_1$: What is the performance of vulnerability prediction models validated using a release-based approach when compared to models evaluated using cross-validation?

We can answer this question by looking at each individual plot and reasoning about the motivations of the observed performance. When considering the dataset as-is, with reference to Figure 6.1, we note that the performance obtained in the release-based validation method is better than the one output by cross-validation. We think that this is due to the high imbalance of the dataset and the different sizes of the considered training and test sets.

Our higlhy unbalanced dataset contains limited samples of vulnerable elements and plenty of samples of neutral elements. Hence, it is easy for the model to recognize neutral elements: this leads to a high number of True Negatives and a low number of False Negatives. The scarce presence of vulnerable samples in the dataset causes low amounts of Positives predictions in general. This impacts on the computation of the performance indicators, leading, for exaple, to high precision, recall and accuracy. So the resulting overall performance seems quite promising, but, in our opinion, it is biased by the dataset conformation.

It is needed to also reason about the size of the training and test sets used to perform cross-validation and release-based validation, respectively. When using cross-validation, the whole dataset is split into 3 folds, of which 2 are used for training and the other one for testing. So the training-test data ratio is 2:1 and the size of the test set is similar to about 30 releases of the application data. When applying the release-based validation approach, 3

releases are used for training and one is used for testing, thus the training-test data ratio is 3:1 and the size of the test set is much lower. We conjecture that there are fewer chances for the model to make mistakes in the release-based validation setting, so the performance resulting from this evaluation method is better. However, the high imbalance of the dataset leads us to come up with poorly meaningful arguments. Thus, we deeperly focus on analyzing the results obtained when using a balanced dataset.

When considering the undersampled dataset, we observe a big change in the performance, depicted in Figure 6.2. The overall results of cross-validation, described by the dots in the plots, are consistent with the findings of the original study by Walden et al. [37] that we partly replicate and that indeed employs undersampling and cross-validation. We note that the release-based evaluation method outputs very different performance and we think that the cause has to be researched in the size of the training set. In each round of the release-based validation, data relative to three releases of the application are used for training the model, and we remember 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 very little data to train and cannot learn how to discriminate among the classes. This causes a severe decline of the performance. The literature suggests [15] [23] [33] [22] that a precision value around 0.7 is reasonable, but the observed values for release-based validation vary between 0.2 and 0.4, so such models could not be employed in a real-case scenario.

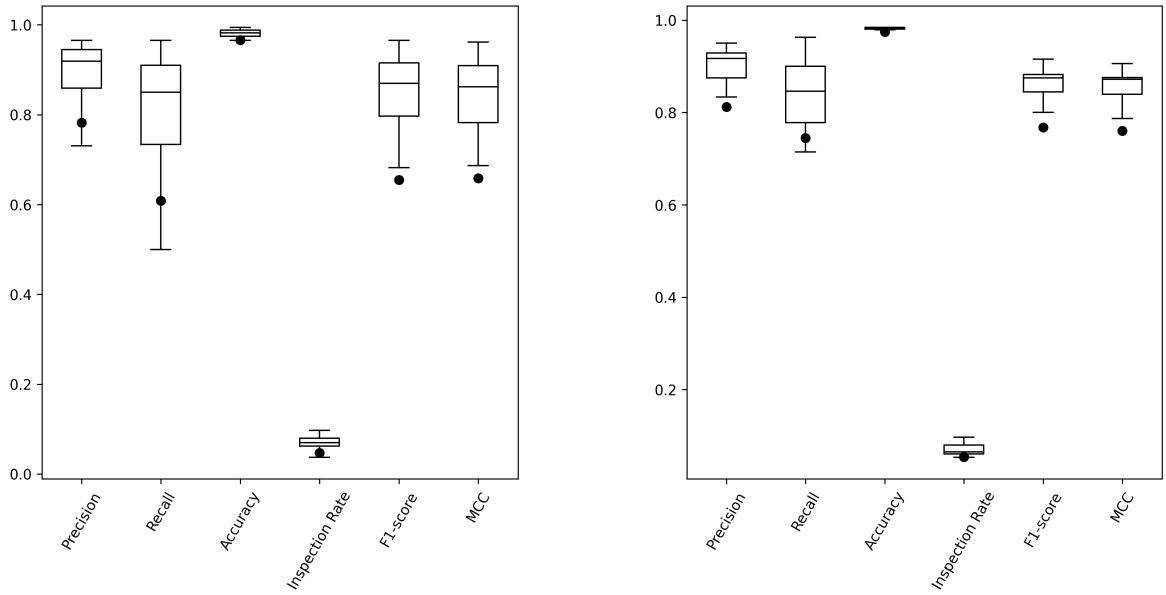
The situation seems to get better when considering the oversampled dataset, as we see in Figure 6.3. The SMOTE oversampling technique is used by Jimenez et al. [15] and allows us to augment the number of vulnerable samples by building synthetic data which is similar to the real one. The model has more opportunity to learn in the training phase, since the dataset is larger and balanced. This results in amazing performance, as we can see by looking at the plots. The cross-validation outputs better indicators, since the whole dataset is used: we remind that in the release-based validation only the data relative to three releases of the application is included in the training set.

**RQ$_2$**: Which modelling approach is more sensitive to the use of a different validation method?

By looking at the side-to-side plots, we can observe the cases in which the box plots relative to the release-based validation method are more distant from the dots representing the performance obtained with cross-validation. We can say that such cases indicate the settings that cause the most difference in the performance. We compute the average differences observed in the two approaches and summarize them in Table 6.2. The average difference in the performance of the Software Metrics based model is higher than the one of the Text Tokens based model, so we assert that the modelling approach based on Software Metrics is more sensitive to the use of a different validation method.

**RQ$_3$**: Which data balancing technique leads to better performance?

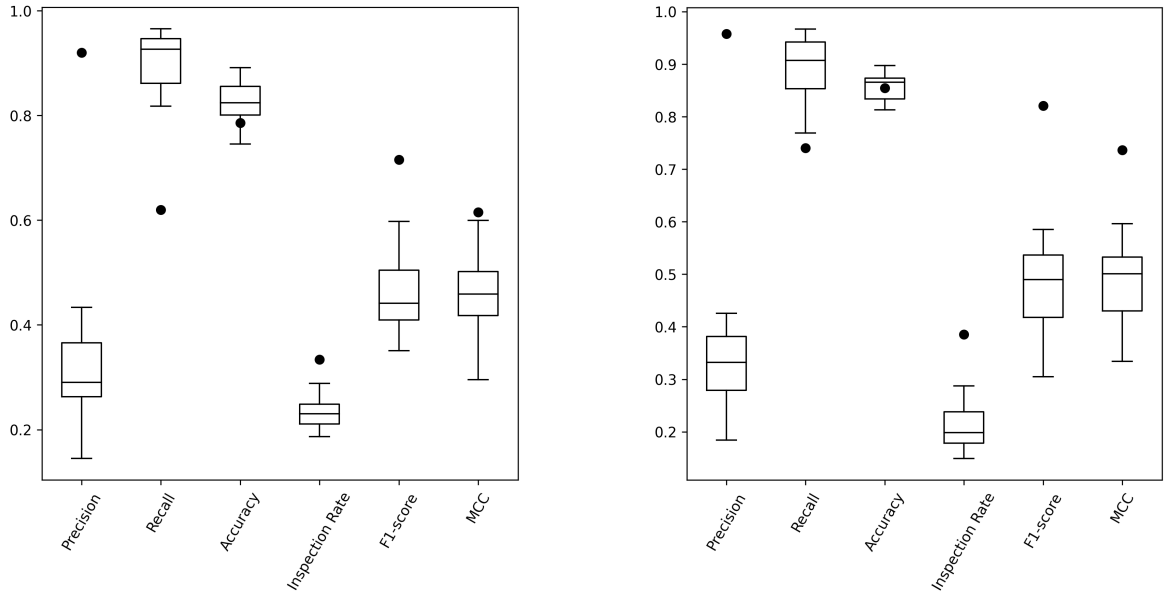
We partially answered to this research question in our previous discussion about RQ$_1$. When performing no data balancing at all, the performance seems to be good, but it is biased by the high imbalance of the dataset. The undersampling technique drastically reduces the size of the dataset that remains available for training, thus resulting in poor performance. The oversampling method seems to be the one leading to the best results.




**(a)** Software Metrics

**(b)** Text Tokens

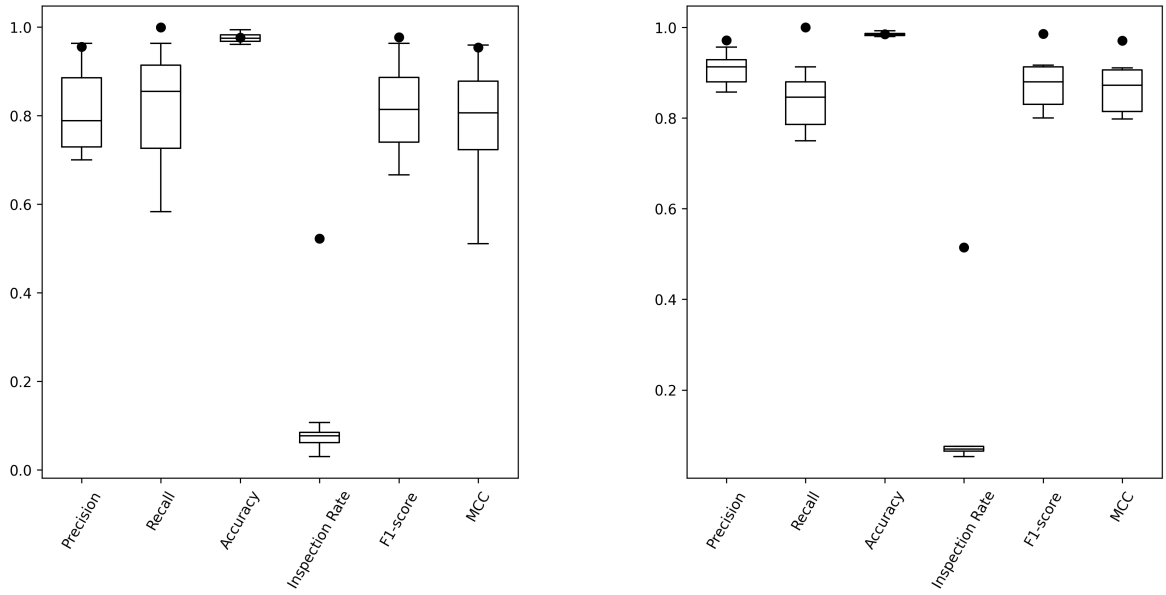
**Figure 6.1:** Performance observed without applying any data balancing technique.




**(a)** Software Metrics

**(b)** Text Tokens

**Figure 6.2:** Performance observed when applying undersampling as the data balancing techinque.




**(a)** Software Metrics

**(b)** Text Tokens

**Figure 6.3:** Performance observed when applying oversampling as the data balancing techinque.

| Performance Indicator | Software Metrics | Text Tokens | Average Difference |
|---|---|---|---|
| Precision | 31% | 26% | 28.5% |
| Recall | 23% | 13% | 18% |
| Accuracy | 2% | 0.4% | 1.2% |
| Inspection Rate | 19% | 21% | 20% |
| F1-score | 23% | 19% | 21% |
| MCC | 18% | 15% | 16.5% |
| Average | 19% | 15% | 17% |

**Table 6.2:** Difference between the performance observed when using cross-validation and the performance obtained when employing release-based validation.

# CHAPTER 7

# Conclusion

In this chapter we draw up a summary of our work and propose our ideas for further research.

We presented our empirical study, whose idea arised from the considerations made by Jimenez et al. [15]. They pointed out that most of the vulnerability prediction models that have been proposed so far have been evaluated using k-fold cross-validation. This technique consists in splitting the dataset in k equally large folds and performing k rounds of validation. In each round, k-1 folds are used as the training set, from which the model learns, and one fold is used as the test set, to evaluate the learning ability of the model. The overall performance of the model is determined as the average of the performance output in the k rounds. Despite being broadly employed, this approach is not realistic if we truly consider the context in which a vulnerability prediction model is supposed to operate. In a real-case scenario, one would be interested in training the model by using all and only the information about vulnerabilities that are known at the training time, and make predictions on those portions of source code whose vulnerabilities are still unknown. Hence, most of the rounds of cross-validation are senseless, since the dataset splitting leads to cases in which the model is trained on newer data and tested on previous data. We wanted to investigate this issue and seize the suggestions made by Jimenez et al. [15].

We performed a comparative study in order to analyze how the performance of vulnerability prediction models is impacted by the choice of an evaluation method over another. We conducted our research using the dataset [36] and the models proposed by Walden et al. [37]. The dataset we selected contains information relative to 75 vulnerabilities found in 95 releases of the PHPMyAdmin application. The prediction models we employed are the one based on software metrics and the one based on text tokens that Walden et al. presented in 2014 [37] and that have been widely investigated since then [10] [35]. We performed our analysis as follows. First, we evaluated the performance of the models by using 3-fold cross-validation. Then, we employed a release-based validation method which consisted in several rounds. At each round, only the data relative to a single release of the application was used for testing, and the three closest prior releases were used for training. We also compared the influence of different data balancing techniques on the performance of the models. We used the dataset as-is, with its high imbalance, then we employed undersampling and then SMOTE [5] oversampling.

We run a total number of 555 experiments and we analyzed the performance observed in all the executions. We noted that the imbalance and the size of the dataset have an high impact on the performance in general. A small dataset is not sufficient for the model to learn, and an unbalanced one makes it difficult to discriminate between classes. We observed from the execution of the experiments that employing the release-based validation method leads to generally lower performance with respect to the cross-validation method. This is consistent with Jimenez et al.'s considerations [15] about the vulnerability prediction models' suitability to be employed in a real-case scenario. In particular, when using an undersampled dataset and applying release-based validation, precision drops from around 0.9 to about 0.3. This value is much lower than the desirable performance of 0.7 that has been suggested to be reasonable in the literature [23] [33] [22].

In this work we investigated the issue of focusing on a release-based validation instead of cross-validating the prediction models, and the replication package is publicly available [30].

In the immediate future we plan to continue researching in this direction, by following Jimenez et al.'s study [15] about the importance of accounting for *real-world labelling* of the training data when applying release-based validation methods. The concept of *real-world labelling* is the following: if we consider to train our vulnerability prediction model at time $t$, we must include in the dataset only those vulnerabilities that have already been discovered before $t$. This makes the empirical analysis context more similar to the real-world scenarios in which the model is supposed to be used. We want to rearrange the dataset that we used in this study, in order to adapt it to this kind of further investigation. A way to do this would consist in tracing back each vulnerability in the dataset to the time it was discovered and reported. After retrieving this information, we could perform a deeper and more realistic analysis on how vulnerability prediction models would perform in a practical context. Another way of obtaining the dataset necessary for the further investigation would be using the Jimenez et al.'s *VulData7* tool [13] [12]. This solution can automatically retrieve the up-to-date vulnerability information relative to a specific software project, but at the moment it is not available. This is due to the fact that the data retrieval mechanism is based on the `NVD XML Data Feed` that has been recently replaced with a `JSON Feed` [24]. We want to collaborate

with the authors of the tool in order to update it and put it into operation again. This would be a great benefit for all the research communitiy. Once the tool is usable again, we also plan to deepen our study by expanding the dataset to other applications.

In this work, we focused on the Random Forest classifier. In the future, we want to analyze the performance of additional algorithms, such as Support Vector Machines and Naive Bayes. We are also curious about the Deep Learning techniques, which are recently getting increasing attention [18] [4]. We point out that if these vulnerability prediction methods do not show adequate performance in a real-world experimental setting, there will be the need to come up with novel approaches in order to support developers in assessing the security of software products, and we will dedicate to work in this direction.

# Bibliography


[1] Phpmyadmin. "https://www.phpmyadmin.net/".

[2] Scopus. "https://www.scopus.com/".

[3] Cagatay Catal, Akhan Akbulut, Ecem Ekenoglu, and Meltem Alemdaroglu. Development of a software vulnerability prediction web service based on artificial neural networks. pages 59–67, 10 2017.

[4] Cagatay Catal, Akhan Akbulut, Sašo Karakatič, Miha Pavlinek, and Vili Podgorelec. Can we predict software vulnerability with deep neural network? 10 2016.

[5] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, Jun 2002.

[6] Xiang Chen, Yingquan Zhao, Zhanqi Cui, Guozhu Meng, Yang Liu, and Zan Wang. Large-scale empirical studies on effort-aware security vulnerability prediction methods. *IEEE Transactions on Reliability*, 69(1):70–87, 2020.

[7] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley, 2007.

[8] The OWASP Foundation. Source code analysis tools. "https://owasp.org/www-community/Source_Code_Analysis_Tools".

[9] Aayush Garg, Renzo Degiovanni, Matthieu Jimenez, Maxime Cordy, Mike Papadakis, and Yves Le Traon. Learning to predict vulnerabilities from vulnerability-fixes: A machine translation approach, 2020.

[10] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.*, 50(4), August 2017.

[11] Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, MetriSec '12, page 7–10, New York, NY, USA, 2012. Association for Computing Machinery.

[12] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. Data7 project: An automatically generated vulnerability dataset. "`https://github.com/electricalwind/data7`".

[13] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. [engineering paper] enabling the continuous analysis of security vulnerabilities with vuldata7. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 56–61, 2018.

[14] Matthieu Jimenez, Mike Papadakis, and Yves 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)*, pages 1–10, 2016.

[15] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark 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, page 695–705, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Aydin Kaya, Ali Seydi Keceli, Cagatay Catal, and Bedir Tekinerdogan. The impact of feature types, classifiers, and data balancing techniques on software vulnerability prediction models. *Journal of Software: Evolution and Process*, 31(9):e2164, 2019. e2164 smr.2164.

[17] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[18] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software

vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[19] Andrew Meneely and Laurie Williams. Secure open source collaboration: An empirical study of linus' law. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 453–462, New York, NY, USA, 2009. Association for Computing Machinery.

[20] Andrew Meneely and Laurie 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*, ESEM '10, New York, NY, USA, 2010. Association for Computing Machinery.

[21] T.M. Mitchell. *Machine Learning*. McGraw-Hill international editions - computer science series. McGraw-Hill Education, 1997.

[22] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, HotSoS '15, New York, NY, USA, 2015. Association for Computing Machinery.

[23] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. pages 529–540, 01 2007.

[24] National Institute of Standards and Technology. Xml vulnerability feed retirement update. "`https://nvd.nist.gov/General/News/XML-Vulnerability-Feed-Retirement`".

[25] Eric S. Raymond and Tim O'Reilly. *The Cathedral and the Bazaar*. O'Reilly amp; Associates, Inc., USA, 1st edition, 1999.

[26] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, January 2000.

[27] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[28] Riccardo Scandariato and James Walden. Predicting vulnerable classes in an android application. In *Proceedings of the 4th International Workshop on Security Measurements and*

*Metrics*, MetriSec '12, page 11–16, New York, NY, USA, 2012. Association for Computing Machinery.

[29] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[30] Giulia Sellitto. The impact of release-based validation on software vulnerability prediction models: Replication package. "`https://github.com/giuliasellitto7/Master-Thesis`".

[31] Giulia Sellitto and Filomena Ferrucci. The impact of release-based training on software vulnerability prediction models. Submitted to ACM WomENcourage 2021. Replication package available at `https://github.com/giuliasellitto7/Release-based-Vulnerability-Prediction`.

[32] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.*, 37(6):772–787, November 2011.

[33] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18, 02 2011.

[34] Kazi Zakia Sultana, Vaibhav Anu, and Tai-Yin Chong. Using software metrics for predicting vulnerable classes and methods in java projects: A machine learning approach. *Journal of Software: Evolution and Process*, 33(3):e2303, 2021. e2303 smr.2303.

[35] Christopher Theisen and Laurie Williams. Better together: Comparing vulnerability prediction models. *Information and Software Technology*, 119:106204, 2020.

[36] James Walden, Jeff Stuckman, and Riccardo Scandariato. Php security vulnerability dataset. "`https://seam.cs.umd.edu/webvuldata/`", 2014.

[37] James Walden, Jeff Stuckman, and Riccardo Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 23–33, Nov 2014.

[38] Wikipedia. Wannacry ransomware attack. "`https://en.wikipedia.org/wiki/WannaCry_ransomware_attack`".

[39] Yichen Xie, Mayur Naik, Brian Hackett, and Alex Aiken. Soundness and its role in bug detection systems. In *Proc. of the Workshop on the Evaluation of Software Defect Detection Tools*, volume 7, 2005.

[40] Yun Zhang, David Lo, Xin Xia, Bowen Xu, Jianling Sun, and Shanping Li. Combining software metrics and text features for vulnerable file prediction. In *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 40–49, 2015.

# Acknowledgements

Throughout my work for this thesis I have received a great deal of support and assistance. I would first like to thank my supervisor, Professor Filomena Ferrucci, whose expertise helped me formulating the research goal and inspired me to perform an interesting study. Your insightful feedback guided me throughout my journey and I am extremely grateful for the precious support that you always showed to me. I hope I can continue drawing my path under your advice.

I would like to acknowledge the SE@SA Lab Research Group. You provided me with the tools that I needed to succesfully complete my thesis.

To my colleagues from the Internet of Things study track go my heartfelt thanks. We walked together for two years and I am sincerely happy for the days we spent together.

All of what I have experienced would not have been possible without the support of my family. You are always there for me and I know I can always count on your love and encouragement.

To my beloved friends Eleonora and Maria Grazia go endless thanks. You showed me what real friendship is and I am immensely grateful to have you in my life. These crazy years in the pandemic have set us apart, but you have always made me feel your love and support, despite the physical distance.

I would also like to thank my boyfriend Gennaro, who has been standing by me during the most difficult moments and helped me regaining my smile.

Finally, my thanks go to all of those I did not mention explicitly, but surely know they have a place in my heart.