Project Report - Inside the Software Vulnerability Prediction Models

Gilberto Recupito
Università degli Studi di Salerno
Email: g.recupito@studenti.unisa.it

Dario Di Dario
Università degli Studi di Salerno
Email: d.didario@studenti.unisa.it

Abstract

Software vulnerability is a particular type of software defect that defines a weakness in the system that can be exploited by an attacker to change its functioning. An attack on the system can have considerable consequences for the system in terms of damage and costs. Vulnerabilities also arise during the software development phase, as certain types of systems are really complex. Although the vulnerabilities are different from defects, they can be defined as a subset of the latter and it's not possible to affirm that a software system doesn't have vulnerabilities. For this reason software solutions for recognize them are born, obtaining a general view of the project state, so than it is possible to act and try to resolve the problem. These tools are generally called Vulnerability Prediction Model (VPM). The VPMs are an approach for prioritizing security inspection and testing to find and fix vulnerabilities. Vulnerability Prediction Models have been created by different kind of metrics and approaches in order to obtain better result trying to know which VPMs have a strong prediction and which have low data requirements. We performed our replications of rebuilding existing VPMs with different open-source project in order to have heterogeneous data with more than 100.000 lines of source code. We then combined the features obtained from single replication and reran the classifier. Finally, we created a performance benchmark to highlight the differences between the results of our replications and the previous experiments, obtaining better results in terms of F-Measure, Recall and Precision.

Keywords

Vulnerability, Software Engineering, Prediction Models, Metrics, Security, Software, Open Source, VPM, Precision, Recall, Accuracy, Benchmark.

1 Introduction

A software vulnerability can be defined as a weakness in the software system that can be exploited by an attacker in order to change the behaviour of the system. Because the number of software systems increases everyday also the number of vulnerabilities increases. An attacker can use vulnerabilities to access into the system and he might take control to damage it,- as launching new attacks or obtaining some privileged information that he can use for his own benefit. Considering this, it is important to know the different types of vulnerabilities, their prevention and detection in order to try to avoid their presence in the final software version of the system and then reduce the possibility of attacks and costly damages. Discovering this kind of issues can be useful in order to evaluate and have a better inspection of the source code components. A manual source code inspection requires human-effort in terms of components/time due to the difficulty to find such vulnerabilities.

Vulnerability prediction models (VPM) are believed to hold promise for providing software engineers guidance on where to prioritize precious verification resources to search for vulnerabilities. There are different type of VPMs, in which each of them have a different ways to obtain source code informations in order to build VPMs that cover different aspects of a software. Finding it permits to discover what are the main features to recognize if a component is vulnerable or not. In this study we have replicated the VPMs created by Shin et al [6] for software metrics, Scandariato et. al [7] for text mining and Gegick et al. [2] for the Automated Static Analysis Code (ASA) with a dataset that includes several open source projects in order to test the VPMs with the heterogeneity of the data. Every single approach can be combined between them to improve the technique accuracy.
With this, we propose the following Research questions:

- **RQ1:** Which are the best VPM analyzed?
- **RQ2:** Which classifier obtains a higher error rate and which one a higher accuracy rate?
- **RQ3:** Is it possible to combine different VPMs? In respect of the state-of-art, are they more effective?

The goal of our study is to continue the contribution given by the previous researcher creating and improving the replication of VPMs of the state-of-the-art and compare them in order to better understand the effectiveness of our replica-

tions to discover vulnerabilities.

The rest of the paper is organized as follow: in the Section 2 discusses background and related work, the Section 3 describe the implementation for each VPM, the Section 4 shows the obtained results and the Section 5 describe the limitation and future works.

2 Related work

In this section, we describe the related work to our comparison.

A vulnerability is an instance of error in the specification, development or configuration of a software such that its execution can be violated by the security policy (implicit or explicit) included in all a company's software.It is considered a weakness of the system which allows an external attacker to reduce the security of information in the system. An exploit means to take advantage of something for one's own end, especially unethically or unjustifiably. An exploit is a piece of software that takes advantage of a bug or vulnerability in order to cause unintended behavior to occur on computer software or hardware. Every vulnerability has a lifecycle as follow:

1. **Discovery**: when a vulnerability is discovered by a seller, a hacker or others.
2. **Disclosure**: the vulnerability is publicly disclosed, that is, everyone will be informed about the vulnerability.
3. **Exploitation**: an external attacker is capable of using the vulnerability to cause problems.
4. **Patching**: when a vendor resolves the vulnerability.

The Common Vulnerabilities and exposure (CVE) is the defacto standard dictionary that provides definitions for publicly disclosed cybersecurity vulnerabilities and exposures. Currently, the MITRE Corporation maintains CVE: this is a nonprofit organization that operates research and development centers funded by all major US federal governments.CVE aims at standardizing the names for all publicly known vulnerabilities and security exposures. The goal of CVE is to make it easier to share data across separate vulnerable databases and security tools. Researchers have used properties of software to prediction vulnerable code inside software project. Theisen et al [3] performed VPMs replication on Mozilla Firefox with 28,750 source code files featuring 271 vulnerabilities using software metrics, text mining, and crash data. Creating a combination of features from each VPM and finally getting results from the classifiers re-runs.

Zimmermann et al. [1] developed a vulnerability prediction model for Windows Vista based on traditional software engineers metrics, such as code churn and number of developers. Gegick et al. [2] built a prediction model using the results of the Automated Static Analysis tool "Flexelint" and discovered that, associated at particular software metrics, it can be used to predict vulnerable components.

Scandariato et al. [7] presented a VPM based on machine learning using the text mining approach,so analyzing directly the source code instead of software or developer metrics. They used an exploratory validation of 20 Android applications and discovered that text mining approach could obtain a prediction power that is equal or superior in relation to other techniques.

Shin et al [6] explored whether fault prediction systems could be adapted for vulnerability prediction, and found that fault prediction perform similarity to specialized vulnerability predictors. Papadakis et al. [8] conducted an experiment based on the replication and comparison of three VPMs in the context of Linux Kernel: import and function calls, software metrics and text mining. They discovered that text mining is best technique when aiming at random instances.

3 Vulnerability Prediction Models

In this section, we describe the three VPMs that we will comparing in this study.

3.1 Software Metrics - Churn and Complexity

Shin et al [6] uses churn and complexity features in order to discover components inside the project's codebase that are likely to be vulnerable. Shin et al. were able to reduce the amount of code inspected for security effort by 71% for Mozilla Firefox, they report the 12% precision and 83% recall using similar metrics. Shin et al. explored execution complexity metrics versus static complexity metrics, and found that execution complexity metrics outperformed their counterpart for Mozilla Firefox and Wireshark in terms of File Inspection Reduction [].

3.2 Text Mining

Scandariato et al. [7] starting from some Java files that includes comments (in-line comments and block-comments). Each Java file is tokenized into a vector of terms, also called *"monograms"* and the frequency of each terms in the file is counted. The frequencies are not normalized to the length of the file due to possible deterioration of performance. The routine used for the tokenization uses a set of delimiters that includes white space, Java punctuation characters and both mathematical logic operators. Scandariato and Walden [7] [9] compared text mining with software metrics approaches to vulnerability prediction. They found that text mining tokens resulted in better precision and recall for vulnerability prediction for three projects: Drupal, PHPMyAdmin and Moodle. Later studies on text mining used author- validated vulnerability sets [9]. Additionally, text mining takes a significant amount of time to run and has a large disk space requirement. The time taken to tokenize code into features is of concern, as it means that a text mining approach to vulnerability prediction would be incompatible with a continuous deployment workflow.

3.3 Automated Static Analysis

Gegick et al. [2] used Automated Static analysis(ASA) tool results as one of the metrics. A static analysis tool is used to analyze the code of a software in order to find defect

without executing the code. The output of an ASA tool is an alert, a notification of a potential fault identified in the source code. This technique provide an early,automated and repeatable analysis to detect faults, but provide as results a high False positive alerts. Gegick, discovered that ASA results has no resolution to determine if a component is vulnerable or not, but in combination with other techniques it can provide better results.

4 Dataset

In this section we discuss the dataset for this experiment.

Sabetta et al. [10] created a dataset of vulnerabilities of open-source software collecting the vulnerabilities from the National Vulnerability Database (NVD) and from project's Web resources that the authors monitored on a continuous basis. The dataset consist of a set in 4-tuples:

(vulnerability_id, repository_url, commit_hash, class)

- **vulnerability_id**:is the identifier of a vulnerability that is fixed in the commit.
- **repository_url**: url of the repository
- **commit_hash**: hash value that identify the commit.
- **class**: this value establish if a commit fix a vulnerability (has *"pos"* value) or it's likely not vulnerable (has *"neg"*).

The dataset covers 205 distinct Java project, and includes 1282 unique commits corresponding to the fixes to 624 vulnerabilities. In addition, this dataset is different from other ones. Sabetta et al. includes in the dataset different vulnerabilities that are not available on the NVD database. There are 29 commit with no CVE identifier and 46 vulnerabilities which have been given a CVE identifier by a CVE numbering authority, but are not yet published on NVD.

5 Methodology

In this section , we describe our methodology about the replication and the comparison of VPM.

5.1 Mining Software Repository

For the extraction of the components containing vulnerability from the dataset, we used Pydriller [11] [12] in order to obtain information regarding the repositories contained in the dataset.

The dataset given by Sabetta et al. [10] contains a hash value that identify the *commit_fix*, and it permit us to reach the modified classes considering the BFIM (before image) and the classes that are introduced in that commit. In MSR theory, the BFIM is defined as the instant in which a file is modified or created before the *commit_fix*. Therefore, we use the before image in order to obtain the instance containing the vulnerability. Our data extraction processing is considering only Java projects, because this filter is already done in the context of the dataset. After that, we have collected and organized in specific folders the several repositories and its commit.

However, there have been some problems with using pydrillers which are: incomplete hash commit, non-existent repository url and memory overhead during the MSR phase. The incomplete hash was handled manually, verifying the existence of the repository via web browser - since the latter require a minimum number of hash commit characters to perform the search, instead PyDriller requires the whole commit - and looking for the commit hash in the history commit of the repository to which it corresponded partially and then modify it. The problem with the non-existence of the project repository - and therefore its commit hash - was handled using the git APIs. In addition, a log file was created that tracks all non-existent repositories. Last, but not least, the overhead problem derives directly from PyDriller. The latter, during the MSR, copied the entire repository contained in the dataset (except for the non-existent ones) to a local workspace, causing a filling of the hard disk memory space without consequent emptying. The problem was handled by opening an issue on the PyDriller repository, notifying the problem.

5.2 Software Metrics Extraction

The process of software metrics extraction of each java file is done through Understand. In line with the experiment done by Theisen et al. [3], we have collected 9 metrics as shown in Table 1. These metrics are initially collected for each component of the projects, in different granularities. Morrison et al. [5] performed a replication of VPMs on two granularity levels: binary level and file level. The results of this experiment showed that the prediction at source level is actionable. Therefore, from all the instances of metrics of granularities extracted by the Understand tool, we selected the file level granularities.

5.3 Text Mining

The realization of the Text mining gave a first approach that respects the state of the art criteria, as described in Section ref sec: Section2. The aim is to work with Java classes and to tokenize words in "monogram" sets, where each monogram will have an associated counter that refers to the number of occurrences found. Below an example of a java class:

```
public class Example(){
/* I'm a
block comment
*/
    public void doRetrieve () {
        //in-line comment
        System.out.println("Hello Guys");
    }
}
```

Before tokenizing it is necessary to remove in-line comments and block-comments, all logical operators and any constants. After tokenization, a result turns out to be as

Table 1. Software metrics used for our replication

Name	Description	Understand_Name
CountLineCode	Number of lines containing source code. [aka LOC]	CountLineCode
CountDeclClass	Number of declared classes in the source code file.	CountDeclClass
CountDeclFunction	Number of Declared functions in the source code file.	CountDeclFunction
CountLineCodeDecl	Number of lines containing declarative source code.	CountLineCodeDecl
SumEssential	Sum of essential complexity of all nested functions or methods.	SumEssential
SumCyclomaticStrict	Sum of strict cyclomatic complexity of all nested functions or methods.	SumCyclomaticStrict
MaxEssential	Max of essential complexity of all nested functions or methods.	MaxEssential
MaxCyclomaticStrict	Maximum strict cyclomatic complexity of nested functions or methods.	MaxCyclomaticStrict
MaxNesting	Maximum nesting level of control constructs.	MaxNesting

follows:

{ *public: 2, class: 1, void: 1, Example: 1, doRetrieve: 1, System: 1, out: 1, println: 1* }

Using a dataset with different projects, this type of execution leads to a construction of a dictionary with a huge quantity of words, increasing the execution times and taking up a lot of disk space. So, we got the strengths of this first approach and thought of applying it in a different way, which makes execution times fast and takes up less disk space. Both quantities are therefore directly proportional. We used a standard normalization process which aims to carry out further "pre-processing" phases before tokenization:

1. **Split CamelCase**
2. **Lower case reduction**
3. **Removing special chars and programming keywords**

These further steps have been applied to the file obtained from the execution of the first approach, obtaining significant improvements in terms of execution speed (less than a minute for all repositories) and in terms of disk space. The approach described by Scandariato et al. Cite Scandariato created a file of approximately 1.20GB, and with approximately 254,500 different words. By applying our preprocessing phases, we reduced the file by 90 % obtaining the file size about 125MB with 13.780 different words. This makes us think that the three phases of "text normalization" are fundamental for finding more occurrences of the same words.

5.4 Automated Static Analysis(ASA) Extraction

The collection of ASA alerts is done by SonarQube. To export the results of the static analysis we installed the plugin "CNESREPORT". Initially the report resulting by SonarQube contained informations about all of alert types: code smells, security hotspots, bugs and vulnerabilities. Due to the number of all these types, we have created a new quality profile in order to execute an analysis based only on vulnerability alerts. Therefore, we performed the analysis on the 39 rules available by the tool (excluding the deprecated rules). From this rules, we studied the results to understand which rules are essential to identify the vulnerability of our dataset, looking for the rules that aren't violated in all java file of our dataset, then we excluded them. The analysis results that 19 types of vulnerability are present in the several projects of the dataset. We have created a new dataset that contains the number of vulnerability of that rule for each file analyzed. Let's define S as the set of files and R as the set of vulnerability. We define $M[i, j]$ with $i \subset S$ and $j \subset R$ in which N is the number of vulnerabilities per class:

$$M[i,j] = \begin{cases} N, & \text{if there are vulnerabilities of rule } j \\ & \text{in the class } i. \\ 0, & \text{otherwise.} \end{cases}$$

The resulting dataset presents 1251 files that violates the 19 rules considered.

5.5 Modelling

To execute the classification with these models, we used Weka. In line with the VPMs presented in the previous experiments, we used the following classifiers:

- Logistic Regression
- Support Vector Machine (SVM)
- Naive Bayes
- Random Forest

The selection of these Machine Learning Algorithms for the classification is due to the goal of obtaining a benchmark of the VPMs replication in the state-of-the-art, highlighting the best algorithm in terms of performance and considering the measurements defined in 5.7

For each selected model, we performed a k-fold cross-validation to use a part of the dataset to fit the model and the other part to test it. In our replication we use five-fold cross-validation, having 5 components, one for the test set and four for the training. In relation of the dataset, five has shown to be a good value for the number of folds of cross-

validation.**??**

5.6 Combination of the features

We have created different combinations of techniques, each one will be evaluated with the classifiers specified in 5.5. Referring to the combinatorial calculation, we define as P_n the number of permutations without repetitions, on the replicated techniques of n. $P_n = n! = 6$ different permutation techniques - in which the single techniques described in the Section 5 are also considered - which are:

- Software Metrics (SM)
- Text Mining (TM)
- Automated Static Analysis (ASA)
- Software Metrics and Text Mining
- Text Mining and Automated Static Analysis
- Software Metrics and Automated Static Analysis

Given the heterogeneity of the datasets deriving from the individual techniques for the number of tuples and for the type of attributes, it was necessary to make an ad-hoc association by creating a unique *id* for each instance in the dataset, formed by *"commit_hash/filename.java"* .

Subsequently it was possible to create a dynamic combination in the following way:
Define D_1 (with a_1, a_2, a_n the set of attributes in D_1) and D_2 (with b_1, b_2, b_n the set of attributes in D_2) two datasets of two techniques studied. The combination of the two techniques will be determined by D where each $d[i] \in D$ is defined:
$d[i] = \{d_1[i], d_2[i]\}$
where $d_1[i]$ is the tuple of D_1 with id i and $d_2[i]$ is the tuple of $D2$ with id i.

5.7 VPM Comparison

After the replication of each single methodology explained in Subsection 5.2, Subsection 5.3, Subsection 5.4 we have considered the possibility of making comparisons, so as to verify which technique it was less performing with different types of classifiers and if the combination of multiple techniques could provide better results than the single ones. Each technique was evaluated with the following measurements regarding the performance of the classification:

•*Precision:* Represents the probability that the VPM's declaration of vulnerable code are accurate. It is function of the True Positive (TP) rate and False Positive (FP) rate of vulnerable source code files. Precision is calculated as:

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

•*Recall:* Represents the probability that the VPM find a source code file that contains at least one vulnerability. It is function of the True Positive (TP) and True Negative (TN) rate of vulnerable source code files. Recall is calculated as:

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

•*F-Score:* Represents the geometric mean of precision and recall. Higher indicates better overall accuracy, assuming that precision and recall are weighted equally. F-Score is calculated as:

$$F_1 = \left(\frac{recall^{-1} + precision^{-1}}{2} \right)^{-1}$$

These measures taken together represent the accuracy and the performance of each VPM in our study. We present these measures separately so the strengths of each model can be determined in lieu of a single accuracy figure. For example, one model may have high recall but low precision, indicating that the model has a high false positive rate. Another model may have high precision but low recall, indicating that the model rarely gives false positives but misses many vulnerabilities.

6 Results

In this section, we present the results of our case study on vulnerability prediction models run against the dataset explained in Section 4.

We have used *Weka* in order to evaluate our VPM with different classifier.

6.1 VPM Comparison

RQ1: Which are the best models analyzed?

Table 2. Median precision, recall and F1 score for each vulnerability prediction model, using Random Forest classifier.

VPMs	Precision	Recall	F1
Software Metrics	0,606	0,620	0,603
Text Mining	0,776	0,766	0,754
ASA Results	0,562	0,596	0,561
SM + TM	0,763	0,755	0,742
SM + ASA	0,620	0,632	0,615
TM + ASA	0,776	0,766	0,755
All	0,783	0,772	0,761

Studying the results of the single techniques present, we note that in Table 2, the VPMs use a classifier *Random Forest*. Analyzing the VPMs with this classifier, we note that the Software Metric (calculated with Understand) and Text Mining (with normalization process) show an increase in precision, recall and F-Score compared to those in [3]:

Software Metrics:

Table 3. Median precision, recall and F1 score for each vulnerability prediction model, using Naive-Bayes classifier.

VPMs	Precision	Recall	F1
Software Metrics	0,548	0,490	0,479
Text Mining	0,653	0,626	0,630
ASA Results	0,544	0,604	0,525
SM + TM	0,665	0,640	0,643
SM + ASA	0,552	0,544	0,547
TM + ASA	0,656	0,628	0,631
All	0,654	0,631	0,634

Table 4. Median precision, recall and F1 score for each vulnerability prediction model, using Simple Logistic classifier.

VPMs	Precision	Recall	F1
Software Metrics	0,543	0,598	0,453
Text Mining	0,720	0,709	0,685
ASA Results	0,570	0,620	0,505
SM + TM	0,711	0,701	0,674
SM + ASA	0,566	0,598	0,472
TM + ASA	0,732	0,725	0,709
All	0,727	0,72	0,702

Table 5. Median precision, recall and F1 score for each vulnerability prediction model, using Support Vector Machine classifier.

VPMs	Precision	Recall	F1
Software Metrics	0,530	0,531	0,531
Text Mining	0,746	0,749	0,745
ASA Results	0,439	0,617	0,475
SM + TM	0,757	0,760	0,757
SM + ASA	0,559	0,597	0,467
TM + ASA	0,757	0,759	0,755
All	0,751	0,754	0,751

- Precision: da 0.45 a 0.60
- Recall: da 0.05 a 0.62
- F-Score: da 0.09 a 0.60

Il Text Mining:

- Precision: da 0.47 a 0.77
- Recall: da 0.05 a 0.76
- F-Score: da 0.09 a 0.60

The obtained results by the ASA are not comparable with the state of the art, as the experiments are based on a different application domain, based on the Recursive Partitioning classifier and on a set of homogeneous data, creating a total difference in the context of the experiment. Despite this, our results could be acceptable:
Automated Static Analysis:

- Precision: 0,57
- Recall: 0,62
- F-Score: 0,50

The results in the previous tables show a significant increase in terms of precision, recall and f-score for each classifier analyzed for the text mining technique compared to software metrics and ASA.

We can therefore believe that among single VPMs, the technique with text mining in a context with heterogeneous data, leads to better results than those described in the state of the art. We also believe it is important to apply the text mining normalization process also to a non-heterogeneous application domain, in order to assert its strengths with certainty.

6.2 VPM Accuracy rate and Error rate

RQ2: Which classifier obtains a higher error rate and which one a higher accuracy rate?

For this research question it is necessary to consider the Accuracy rate and Error rate, using the formulas:

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

which indicates the accuracy of the model used. The best accuracy assumes a value of 1, while the worst is 0.

$$Error = \frac{FP+FN}{TP+TN+FP+FN} \text{ o } Error = 1-Accuracy$$

which is calculated as the number of all incorrect predictions, divided by the total number of the data set. On the opposite here, the closer you get to 0, the lower the margin of error committed by the classifier, the closer you get to 1 the greater the margin of error.

Referring to Table 6, we calculated the accuracy for each VPM with the respective classifiers, then we calculated an average of accuracy for each classifier, in order to establish the lowest performing one:

1. *Random Forest* : 0,718
2. *Support Vector Machine* : 0,690
3. *Logistic Regression* : 0,674
4. *Naive-Bayes* : 0,592

therefore, we identify that the *Random Forest* obtains a higher accuracy value, while the *Naive-Bayes* the lowest.

6.3 VPM Combination

RQ3: Is it possible to combine different VPMs? In respect of the state-of-art, are they more effective?

Table 6. Accuracy for each vulnerability prediction model and classifier.

VPMs	RF	NB	LR	SVM
Software Metrics	0,619	0,490	0,598	0,531
Text Mining	0,766	0,626	0,709	0,748
ASA Results	0,595	0,603	0,619	0,617
SM + TM	0,754	0,639	0,700	0,759
SM + ASA	0,632	0,543	0,597	0,596
TM + ASA	0,766	0,627	0,725	0,758
All	0,772	0,630	0,719	0,753

The text mining technique return good results regardless the type of the classifier, in fact the implementations of the VPMs combined with the text mining affects positively on the obtained data, compared to other combined VPMs without text mining. Note, in fact, that for each classifier the combined techniques with the text mining get better results than this single technique. Moreover, the results in the previous tables show that it's not always possible to obtain improvements from the combination of the VPMs, but even in some cases, the addition of features can decrease the performance of the model. We consider important to do a study of the single techniques in order to choose accurately the types of combinations. Therefore, it turns out possible to combine the VPMs to improve the performance of a model, and, even if it's not possible to retrieve in the state-of-the-art all the combinations, we consider that the combination of the Text Mining and Software Metrics obtains better results than those in [3], having an increase of Precision, Recall and F-Score with all the described classifiers. With these results, we can say that our models are not only more precise (Precision of 0.76), but even more sensitive (Recall of 0.76).

7 Conclusion and Future Works

In this paper we present the replications of the Vulnerability Prediction Models in the state-of-the-art.

These replications are performed on a cross-project dataset, which contains several types of vulnerabilities for each project. Moreover, every single project in this dataset is open-source. The approaches presented from the replications analyzes the source code through text mining, the software metrics and the alerts of the automated static analysis.

The results of this study show that the text mining is the best technique for predicting vulnerable components e in general every single technique perform better results than those of the state-of-the-art.Subsequently these techniques were combined with each other in order to try to increase the predictive power of the VPM. Nevertheless, while combining the techniques with each other - and obtaining better results than the state of the art - we have noticed that the deviation of prediction from text mining is minimal. This points out that the techniques combined with text mining contribute very little in trying to get better predictions. As a counter-proof, we studied the data obtained from the combination that does not use text mining (Software Metrics and Automated Static Analysis), obtaining in some cases lower results.

In the future, we have planned to work on the development of a general framework for the use of VPM, in which there will be an evaluation and prediction phase, where in the evaluation phase an evaluation of the learning schemes will be obtained and the best.Later in the predicting phase the best learning scheme is used to create a predictor with all historical data and use the latter as a component to predict the vulnerabilities on new data. The first phase of the future general framework relate to evaluation is proposed in this research work. As another future development, a re-engineering process can be provided that allows the use of methodologies as a fundamental tool for predicting vulnerabilities.

In the end, we have planned to perform an information gain technique in order to discover the "amount" of information of the presented VPMs attributes. This technique can be useful for a better understanding of the prediction power of the VPMs.

References

[1] T. Zimmermann, N. Nagappan, and L. Williams, “Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista,” in ICST’10

[2] M. Gegick, P. Rotella, and L. Williams, “Predicting Attack-prone Components,” in ICST’09.

[3] C.Theisen, L.Williams, ”Better togheter: Comparing vulnerability prediction models”, North Carolina USA, 2019

[4] T. Hastie, R. Tibshirani, and J. H. Friedman, The Elements of Statistical Learning, New York, Springer, 2001.

[5] P. Morrison, K. Herzig, B. Murphy, L. Williams, ”Challenges with Applying Vulnerability Prediction Models”.

[6] Y.Shin, A. Meneely, L.Williams, J.A Osborne, ”Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities, IEEE Trans. Softw. Eng. 37 (6) (2011) 772-787.

[7] R.Scandariato, J.Walden, A.Hovsepyan, W.Joose, ”Predicting vulnerable software components via text mining”, IEEE Trans.Softw Eng. 40 (10)(2014) 993-1006.

[8] M. Jimenez, M. Papadakis, Y. Le Traon, ”Vulnerability Prediction Models: A case study on the Linux Kernel”, 16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2016.

[] Y. Shin, L. Williams, An initial study on the use of execution complexity metrics as indicators of software vulnerabilities, in: Proceedings of the 7th International Workshop on Software Engineering for Secure Systems, SESS ’11, ACM, New York, NY, USA, 2011, pp. 1–7, doi:10.1145/1988630.1988632

[9] J. Walden, J. Stuckman, R. Scandariato, Predicting vulnerable components: software metrics vs text mining, in: Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on, IEEE, 2014, pp. 23–33.

[10] S.E. Ponta and H. Plate and A. Sabetta, M. Bezzi , C. Dangremont,”A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software”,Proceedings of the 16th International Conference on Mining Software Repositories

[11] D.Spadini, M.Aniche, A.Bacchelli, ”PyDriller: Python Framework for Mining Software Repositories”,26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE’18)

[12] https://github.com/ishepard/pydriller