Vulnerability Prediction Models: A case study on the Linux Kernel

Matthieu Jimenez, Mike Papadakis and Yves Le Traon
SnT, University of Luxembourg, Luxembourg
firstname.lastname@uni.lu

***Abstract*—To assist the vulnerability identification process, researchers proposed prediction models that highlight (for inspection) the most likely to be vulnerable parts of a system. In this paper we aim at making a reliable replication and comparison of the main vulnerability prediction models. Thus, we seek for determining their effectiveness, i.e., their ability to distinguish between vulnerable and non-vulnerable components, in the context of the Linux Kernel, under different scenarios. To achieve the above-mentioned aims, we mined vulnerabilities reported in the National Vulnerability Database and created a large dataset with all vulnerable components of Linux from 2005 to 2016. Based on this, we then built and evaluated the prediction models. We observe that an approach based on the header files included and on function calls performs best when aiming at future vulnerabilities, while text mining is the best technique when aiming at random instances. We also found that models based on code metrics perform poorly. We show that in the context of the Linux kernel, vulnerability prediction models can be superior to random selection and relatively precise. Thus, we conclude that practitioners have a valuable tool for prioritizing their security inspection efforts.**

I. Introduction

Developing secure software forms a mandatory requirement for almost all software organizations. To ensure secure development, industrials adopt dedicated security life cycle processes which aim at identifying and fixing them [1]. Among the several types of vulnerabilities, code-based ones are responsible for the majority of the exploits [2] (software, data or commands that can lead to a security policy violation [3]), which unfortunately increase every year [4].

Vulnerabilities can be seen as a special kind of defects. Depending on the application, they can be more important than bugs and require a quite different identification process than defects. For instance, seeking for vulnerabilities requires an attacker's mindset [5] to understand and exploit the weaknesses of a system. Usually, defects are (easily) noticed by users or developers during the "normal" operation of the system while vulnerabilities pass unnoticed. There are also many more bugs than vulnerabilities [6] (at least many more bugs are reported every year). Furthermore, vulnerabilities are critical [1], [2], while many bugs are not, i.e., they are never fixed. Finally, most developers have a better understanding of how to identify and deal with defects than with vulnerabilities.

Discovering such issues is a hard and costly procedure [1]. To support this process, researchers have developed vulnerability prediction models, which guide security inspections, such as code reviews [2] or security testing [5], by pointing out components that are likely to be vulnerable [7]. In fact, there are three main approaches, here named as *software metrics* [8], *text mining* [7] and *includes and function calls* [9]. Unfortunately, previous studies do not make a reliable and comprehensive comparison between them. Also, to the author's knowledge there is no replication study as deserved, given the importance of the problem. Therefore, our objective is to replicate and compare vulnerability prediction models.

This paper lies in the context of the Linux kernel. We thoroughly compare the approaches under both "experimental" and "realistic" cases using a much larger and reliable dataset of publicly reported vulnerabilities than any other study. We believe that effective research should be bounded to a particular context since it is likely that 'universal' models fitting on every domain do not exist. In particular, our models involve training on data that are project specific. Previous research has shown that cross project models are generally less effective than the project specific ones [9].

The majority of previous research, i.e., [7], [9], [8], seeks to predict the vulnerable source code files. This choice was confirmed as actionable by Microsoft Windows developers [10] and thus, we make our evaluation at the file granularity level. Another interesting observation from the above-mentioned study was that the balance of a dataset and the way it is formed can influence the accuracy of the prediction models. Therefore, considering this point, we investigate two different settings, named as "experimental" and "realistic". The first one aims at determining whether the studied approaches can discriminate between vulnerable files and likely to be buggy files, while the second one at evaluating the practical effectiveness of the approaches as if they were used by the developers (the proportion of vulnerable files is approximately equal to the proportion reported on the Linux releases).

Our study aims at making a comprehensive evaluation of the studied approaches by reliably setting the training and evaluation datasets. Thus, our investigation is based on two scenarios, one that splits the training and evaluation sets at random and a "practical" one, based on time, where with respect to given reference time points (release times), we train on the past vulnerabilities and to predict (evaluate) the future ones. Overall, the present study involved a set of 1,640 vulnerable files, 54,000 clear files and 870 experiments. To build a reliable 'ground truth', we mined and linked all Linux kernel vulnerabilities reported in the National Vulnerability Database (NVD) between 2005 and 2016.

Our results show the importance of the element selection when building a dataset. Indeed, models can perform better or similarly in a largely unbalanced dataset, which is closer to the reality, than in a balanced dataset composed of similar but distinct elements, i.e., bugs and vulnerability.

Overall, our data suggest that the variations on the studied application scenario can greatly influence the performance of the prediction approaches. In our two scenarios, we find that when aiming at future vulnerabilities, the approach based on includes and function calls is performing better, while when aiming at random instances of vulnerabilities irrespective to time, text mining outperforms the other approaches.

The remainder of the paper is organized as follows. Section II details the scope of the prediction models and the related work. Section III motivates and states the studied research questions. The construction of the used dataset and a description of the replicated methods is given in Sections IV and V. Sections VI and VII respectively present and discuss our results. Finally, Section VIII concludes the paper.

II. Background

A. Vulnerability Prediction Modeling

Vulnerability prediction modeling is a relatively recent field of study which aims at automatically classifying software entities as vulnerable or not. The goal of such an attempt is to support code reviewers by pinpointing the entities to put their efforts. A major benefit of these methods is that they are software agnostic, i.e., they do not require any specific knowledge of the software under analysis. Thus, they can easily be adapted to different software projects by using an appropriate training set.

Vulnerability prediction can be seen as a subfield of defect prediction. A comprehensive presentation of these approaches can be found in a recent survey by Hall et al. [11]. However, as already stated in section I, there are specific aspects that make vulnerabilities a particular kind of defects that require a special treatment.

B. Vulnerable Entity

An entity refers to the piece of code that should be inspected. Thus, depending on the context and the needs, we can choose different entities (or levels of granularity, e.g., statements, methods or modules) to work with. Gegick et al. [12] focused the module level, as it was a goal required by Cisco. While lower levels of granularity might be useful, there is always a trade off between this and the accuracy. Currently, most of the published papers suggest that working at the file level is a good compromise. Furthermore, a recent study [10], reports that this level of granularity was reported as acceptable by developers at Microsoft.

Defining a reliable dataset is not an easy task, since it strongly depends on the information that can be mined from software repositories. For example: Neuhaus et al. [7] and Shin et al. [13] used Mozilla Firefox Security Announcement (MFSA) to build their vulnerability dataset. Other work tried to approximate it using a static analysis tools. For example, Scandariato et al. [9] used Fortify SCA, a static analysis tool to built their dataset. Thus, previous research labels as vulnerable a component that was blamed in a security report or flagged as vulnerable by a tool.

To built our dataset we rely on the Common Vulnerabilities and Exposures (CVE)-NVD [4] database. Thus, we consider as vulnerable the components that were modified to fix vulnerabilities. We use CVE-NVD because it is among the largest vulnerability databases and it contains vulnerabilities that are acknowledged by the Linux developers when they are fixed. Thus, we believe that they form a reliable piece of information. Zimmermann et al. [14] also followed a similar process in their study on Windows Vista.

C. Related Work

Neuhaus et al. [7] was among the first vulnerability prediction approaches. The authors discovered a correlation between import/function calls and vulnerabilities and used it to form a prediction approach, using the includes and function calls of the files under analysis. In other words, the includes and function calls were the features used to train a classifier. Their results on the Mozilla Firefox project showed that a recall of 45% and a precision of 70% can be achieved.

The use of software metrics to build vulnerability prediction models has been attempted in several studies. Shin et al. [8] suggested the use of complexity metrics along with code churn and developer metrics. The authors validated their approach on Mozilla Firefox and Linux Kernel and report a recall of up to 86% for Mozilla Firefox and up to 90% for the Linux Kernel. However, they also reported a low precision.

In a follow up study [13], the same authors tried to figure out whether a traditional defect prediction models, trained on complexity, code churn and past fault history is capable to predict vulnerable components. They also tried to perform training to distinguish between bugs and vulnerabilities and found out that it was hard to do so, as their results were similar for both examined cases.

Chowdhury and Zulkernine [15], [16] proposed another approach based on software metrics but using complexity, coupling and cohesion. They evaluated it on Mozilla Firefox and achieved a means recall of 74.22%.

Zimmerman et al. [14] presented an empirical study on Windows Vista to evaluate the efficacy of code churn, code complexity, dependencies and organizational measures to build a vulnerability prediction model. They obtained good precision but low recall. Nguyen et al. [17] introduced an approach based on dependency graph to train the model and evaluated it on Mozilla Javascript Engine.

In principle, all the above-mentioned approaches are similar. They differ from the application context and the formed evaluation datasets. In the present study, we replicate the approach of Shin et al. [8] because it is well suited for C/C++ programs and it is one of the most popular approaches both for vulnerability and defect prediction. It also uses a large and comprehensive set of metrics which is used by almost all the other complexity-based approaches.

Scandariato et al. [9] suggested using text mining technique to build the prediction model. This is a commonly used technique for natural language processing tasks and it was introduced for defect prediction by Hata et al. [18]. The approach is simple as it decomposes the source code into a bag of words which is then used to train a classifier. This approach was validated on 20 android applications, yielding a precision and recall of approximately 80%. The dataset was built using a static analysis tool, which raises concerns regarding the drawn result since it is widely known that such tools are quite imprecise, hence they produce a lot of type I errors. Another issue is that the prediction model predicts what the tool does, which raises concerns regarding type II errors as well.

Such concerns led to another work, i.e., by Walden et al. [6], which tried to evaluate the same approach on different settings. They used three web applications written in PHP. However, the relatively small size of the datasets they used, i.e., approximatively 30 vulnerabilities per application, brought the author to base their evaluation on cross validation as there was not enough data to create two independent sets. This raise serious concern regarding the validity of the reported results since the used evaluation settings have been shown to lead to generalisation and overfitting problems [19].

By contrast, our study uses a large dataset composed of 743 vulnerabilities that was split into independent training and evaluation data sets. We also closely replicated the complexity metrics approach suggested by Shin et al. [8] and we also compared with the includes and function calls approaches.

The previously introduced approaches are somehow generic and can work on most of the existing software. However, they do not specialise on specific types of vulnerabilities. Jimenez et al. [20] analysed Android vulnerabilities and found that they are of 13 different types. Good news are that they also found that they tend to be on the most complex functions, which suggest that prediction models might be suitable. A specialised approach is the work of Smith et al. [21] on the SQL Hotspot, which found a correlation between vulnerability and the number of SQL statements. Gegick et al. [22] used the warnings of security tools along with complexity metrics to build their prediction models. Similarly, Gegick et al. [23] tried to use non-security failure reports to build their models. These are two examples of approaches requiring additional data and/or the help of a tool.

III. Research Questions

The goal of the present study is to replicate and compare existing vulnerability prediction models. There are three main methods in the literature, i.e., software metrics [8], text mining [9] and includes and function calls [7]. Interestingly, these have never been replicated or compared; every study has been evaluated on different contexts and custom datasets. Thus, the need to deal with their external validity and comparing them on a large and reliable "ground truth" dataset is evident.

Arguably one of the most important questions in predictive modeling is the ability of the developed models to identify, among several elements, the ones that it seeks for. In our context, vulnerability prediction model should be able to distinguish between vulnerable and non-vulnerable files. Among non-vulnerable files some are closer to vulnerabilities, i.e, buggy files. Indeed, vulnerabilities are often referred as security bugs and thus consider as a subgroup of bugs. Hence, it is interesting to determine if the approaches are able to determine this subgroup, i.e., vulnerabilities, or are just pointing toward the upper group.

Thus, our first research question investigates whether the studied approaches can distinguish the vulnerable from buggy files.

RQ1. Are the vulnerability prediction models capable of distinguishing between vulnerable and buggy files?

A positive answer to this question will provide a good indication of whether the studied methods are of any value in our context.

Being able to distinguish vulnerable files does not imply that the model is actually useful for developers. This was investigated by Morrison et al. [10] who found that when a tiny proportion of files is vulnerable, the usefulness of the prediction models is hindered. Therefore, we seek to investigate the effectiveness of the examined approaches under cases that are close to reality, i.e., when the proportion of vulnerable and non-vulnerable files in the studied data set are close to the ratios that are found in the Linux kernel (3% of files have a vulnerability history). Thus, we ask:

RQ2. What is the discriminative power of vulnerability prediction models in distinguishing between vulnerable and non-vulnerable files in a realistic environment?

This question is as an attempt to investigate the actual prediction power of the studied approaches. Evidently, a high (respectively low) accuracy indicates a relatively good (respectively bad) prediction. Also, the difference between the results of RQ2 from those of RQ1 demonstrates the impact of the datasets (proportion of vulnerable files) on the discriminative power of the models.

To address these two research questions, we created two distinct datasets named "experimental" and "realistic". These datasets will be presented in section IV-C.

Although in RQ2 we use what we call the "realistic" dataset, this does not accurately assess the predictability power of the developed models in identifying future vulnerabilities. In other words, we need to investigate the extent to which future vulnerabilities can be captured by the developed prediction models when trained on past data. Therefore, we investigate the ability of the models to capture the vulnerabilities of the different Linux kernel releases using the data of the past releases.

RQ3. How effective are the vulnerability prediction models in predicting future vulnerabilities when using past data?

The answer to this question provides a complete picture regarding the practicality of the approaches. Additionally, a relatively good accuracy on the predictions will indicate that

vulnerabilities share the same characteristics over time, since they are captured by the prediction models.

Up to this point, our discussion is solely based on the effectiveness perspective. However, we have left aside any discussion regarding the cost of each method. This information can be useful for researchers or practitioners dealing with frequently changed data. Thus, our last research question evaluates the time and memory needed to train and develop each of our models.

RQ4. What is the cost in terms of memory and time consumption of building the studied vulnerability prediction models?

IV. Dataset

As already mentioned a large and reliable dataset is mandatory. As the dataset from previous studies were either outdated, small, not available or questionable, we choose to create our own dataset with the latest data available on the Linux kernel. This section explains our choice of the Linux kernel, the vulnerability mining process and the overall procedure for building our datasets.

A. Linux Kernel

Linux kernel is the core of all Linux operating systems. Initiated in 1991 by Linus Torvalds as a hobby, it is now embedded in billions of devices (used by all Linux operating systems and all Android devices). It is also the biggest open-source project with more than 19.5 million lines of code and contributions from over 14,000 developers.

While these numbers are impressive and naturally could ensure the adequacy of the collected data, they are not the only reason for choosing the Linux kernel, as a source to build our dataset. As mentioned earlier, one of the most critical points when building a dataset lies in the choice of the ground truth. This is where the Linux kernel really becomes interesting. Linux development community is well organized and works with strict guidelines[1]. As a result, it has a long and well-reported history of vulnerabilities, which makes it easy to both find many vulnerability reports and map to them with their corresponding vulnerable files. Indeed, about 75% of the vulnerability reports for Linux that we studied contained a link to a patch.

Another point that is worth mentioning is the availability of valuable information from the software repositories. In the past few years, many open source projects switched to the git platform. This makes the links to the previous version control systems as reported in the vulnerability reports invalid. Thus, we lost a major part of the history resulting in a strong impact on both on the size of the datasets and their accuracy. In other words, we need stability in the version control system in order to build a reliable and relatively large dataset. We found that Linux kernel fit this description. Indeed the Linux community created git in 2005 and was therefore the first to adopt it. This means that we cannot find a project with a longer history of using git, i.e., about 10 years.

[1]https://www.kernel.org/doc/Documentation/CodingStyle

B. Linux Vulnerabilities

To build a dataset suited for our study, we must start by collecting as many vulnerable files as we can. To do so, we first have to find a source of vulnerability reports. In this study we chose the CVE-NVD database as Linux kernel vulnerabilities are usually all reported in it (as of today more than 1,300 vulnerabilities have been reported for the Linux kernel since 1999). Furthermore, we focus on vulnerabilities that happened after 2005 (date of adoption of git) as it is no longer possible to retrieve all necessary information for vulnerabilities before this date. This leaves us with 1,050 vulnerability reports to explore. Once all of the reports are gathered, we proceed as follows:

1) We collect all remote repository git URLs, i.e., in the case of the Linux kernel, two remote repositories are available, one at git.kernel and another one available on GitHub. We then create a regular expression to extract the commit hash from the git URLs that point to these repositories.
2) We extract all commit hashes that are present in the URL of the vulnerability reports using the regular expression.
3) To complete this first set of commits, we browse all the commit messages of the Linux kernel git, representing more than 570,000 commits, looking for a reference to a CVE number. In case of matches, we keep the commit hashes.
4) Finally, we retrieve all vulnerable files from the above list. Here, a vulnerable file is a file that is mentioned in the patch commit and existed before the patch. We focus on files written in C as they form the great majority of the collected files. Also, the prediction models studied are not designed for multi-language programs.

We ended up with 1,640 vulnerable files accounting for 743 vulnerabilities.

C. Dataset Building

Once all vulnerable files have been gathered, the next step was to select sets of non-vulnerable files. We built two distinct datasets: one corresponding to RQ1, the experimental dataset, which is composed of a slightly unbalanced set of vulnerable and likely to be buggy files, and another one designed for RQ2, the realistic dataset, which is close to the practical cases (fully unbalanced instead of slighlty). To answer RQ3, we used the two datasets.

1) Experimental Dataset: This first dataset was created to determine whether models were able to distinguish vulnerable files from closely related ones, i.e., buggy but non-vulnerable ones. As there is no real set of buggy files existing, we chose to use as be buggy files for this dataset, files that were present in a commit mentioning Bugzilla reports. It is noted that even if the major part of these files are buggy, some might not. Thus in the following we will refer to those files as "likely to be buggy". To retrieve them, we followed a procedure similar to the one for vulnerabilities: we browsed all commits in the Linux kernel repository looking for a mention to a Bugzilla

report reference and collect the corresponding commit hash (about 3,400). From there we collected all the files present in these commits that were not related to a vulnerability and declared them as non-vulnerable (about 4,900). Overall, the proportion of the vulnerable files of this dataset is 20%.

2) Realistic Dataset: This dataset was designed to investigate whether the models were able to flag vulnerable files under a realistic setting, i.e., in an environment where vulnerable files would be uncommon. This implies the creation of a largely unbalanced dataset. After some analysis, we found out that about 3% of the files in Linux have a history of being vulnerable, 47% of being linked to bug patches and 50% were never impacted by a vulnerability or a bug. Thus to reproduce these proportions, we randomly selected for every vulnerability 31 files that were never declared as vulnerable. These 31 files were mined from the project according to the time that vulnerability was declared, i.e., we ignore the changes that made after this point in time. To be as close to the reality as possible, among these 31 files we randomly select 15 of them from a pool of files that have a history of being linked to a bug patch and the remaining 16 from a pool of files that were never implicated in a patch. Thus, we constructed a set of approximately 52,500 files that reflect the actual ratios of vulnerable, buggy and non-vulnerable files in Linux, i.e., 3% vulnerable, 47% of being linked to bug patches and 50% clear files.

3) Dataset Specificity: Our datasets are automatically generated without any manual addition or removal of files required. They are both easy to update and adapt to any other open source software that has an organized community. They are created using a commit-based approach since we only know when the vulnerabilities were fixed. This approach has the advantage over a release based one of only considering a vulnerability once which can significantly reduce the noise in our dataset. Indeed some vulnerabilities might be present in more than one release, i.e., the time for them to be uncovered.

V. Studied Methods

Replicating a vulnerability prediction study is not always straightforward since several points have to be considered and some decisions taken by the initial study might be unknown. A first point regards the employed measurements as they are realized by the used tools. The tools used in the original study might not work in the new environment or might simply be unavailable. Thus, there is a need for using compatible tools with the same functionality. A second important point concerns the machine learning aspects. To avoid bias we should use the same machine learning techniques and the same filters as those applied in the original study. Yet, filters used are not always clearly indicated in all the studies.

In this section, we describe how we replicated the selected approaches including details on the features extraction and machine learning techniques that we applied. We believe that doing so will allow an easy and precise replication of our results. It will also enable a direct comparison with new, future, approaches.

A. Include and Function Calls

This approach is based on a simple intuition, which is that vulnerable files share similar sets of imports and function calls. Thus, Neuhaus et al. [7] built a model using the profile of either imports or function calls to discriminate between vulnerable files and non-vulnerable ones.

1) Features Extraction: To build such models, we first need to extract for each file under analysis its includes and its function calls. To do so, we used a simple regular expression for gathering the includes and used the file Abstract Syntax Tree (AST) to retrieve the function calls. To generate the AST of a file, we used the JOERN tool[2].

2) Machine Learning Technique: Once all the features have been extracted, we create a list of all features (either include or function calls) that are present in all of the files under analysis of our training set. Machine learning is based on a feature matrix M that uses the feature elements of the feature list as columns and the file under analysis as rows:

$$M_{ij} = \begin{cases} 1 & \text{if file } i \text{ features feature } j, \\ 0 & \text{otherwise.} \end{cases}$$

We also add to M a column indicating whether the files are vulnerable or not. We can then use this matrix as training data for the machine learning algorithms. Neuhaus et al. [7] decided to use a Support Vector Machine (SVM) algorithm with a linear kernel for this task. We used the Weka[3] core library and its libSVM module for the replication.

B. Software Metrics

Shin et al. [8] conducted many studies using software metrics. We replicate their most comprehensive one based on complexity, code churn and developer activity metrics. These metrics were used as an indicator of software vulnerabilities [8], mainly due to their success in defect prediction.

1) Features Extraction: In their work, Shin et al. created 4 models: one for every kind of metrics, and one combining them. We report results for the combined one since this gave the best results. To build such models, we first have to make the metric measurements for the studied files.

Complexity. In the original study, 14 complexity metrics were suggested. These were broken down into three categories, i.e., intra file complexity, coupling and comments density. These metrics were computed using the Understand tool[4]. In our study, we tried to use the same tool, but unfortunately this tool only works through its graphical user interface. This is impractical for the case of the commit-base analysis we performed since it would require a manual collection of thousands of data. We also had technical issues, such as problematic handling of some code parts and crashes, which prohibit its use. Thus, we developed and reimplemented the suggested metrics in a prototype tool, manually tested and automatically compared our results with those of the Understand tool for some cases.

[2]https://github.com/fabsx00/joern
[3]http://www.cs.waikato.ac.nz/ml/weka/
[4]http://www.scitools.com

Code Churn. The original work was suggesting the use of 3 metrics, number of changes, number of changed lines and number of added lines. The way to retrieve these metrics is not described in the original work but was easy to assume. Indeed, we generated our dataset based on commit and thus, we computed these metrics by browsing the git history of the selected files.

Developer Activity Metrics. The initial study used a tool developed by its authors which was not available. Thus, we had to reimplement it. However, in their study Shin et al. discovered that the only interesting metric of this category for vulnerability prediction was the "number of developers who have worked on every component". To minimize the risk of bugs in the implementation as well as the computation cost, we only focused on this one as it can be easily retrieved from the git history. Indeed, the cost of computing a whole developer network for the remaining metrics are important given the size of the Linux kernel.

2) Machine Learning Technique: In their work, Shin et al. hypothesized that a subset of either complexity, code churn and developer activity metrics or a combination of them could predict vulnerable files. To select the right subset for each case, we, as suggested, only keep the three best metrics to build the model using Information Gain as ranking. Regarding machine learning, we used as in the original study logistic regression with Weka.

C. Text Mining

Text mining was suggested for vulnerability prediction by Scandariato et al. [9]. The underlying idea of their study was to suggest a method capable of choosing features without any human intuition. This is in contrast to the other two approaches we replicate. As we stated, this approach creates a bag of words from the source code of the training files under and builds a model based on them.

1) Features Extraction: The feature extraction of this approach is quite simple. The file's source code is split into tokens which are imported to a vector of unigrams. Then, the frequency of each unigram in the file is computed. The delimiters for the tokens are based on the language punctuation characters and the frequency is not normalized. In this replication, we reimplemented the proposed tokenizer by adapting it for the C punctuation.

2) Machine Learning Technique: Once all the features have been extracted, we proceed in a similar manner as the include and function calls by creating a list of all unigrams that are present in all the files of the training set. Then, we built the feature matrix based on it.

To improve the performance of the predictions, Scandariato et al. suggested discretizing the count of each word and making it binary using the method of Kononenko [24]. The discretization would first be computed on the training set and then apply on the testing set. The features rendered useless by the discretization (all binary values are the same in the training set) would then be removed. This part required search to retrieve the right filters with the right options in Weka, similarly to what was used by the authors. In the end, we figured out that the first filter to use was "Discretize" with the options "Kononeko", makes binary and use bin number activated and the second one was "RemoveUseless". The need for these two filters is to reduce the number of features which are exploding (up to 2 million in our dataset). Thus, we found out that these filters were indeed able to divide the number of features by 10. Finally, we used Random Forest with 100 trees for the machine learning part as the authors found that this algorithm was performing better than the other algorithms.

VI. Evaluation

A. Methodology

1) Experiments: For RQ1, we used the experimental dataset and two evaluation methods; stratified 10 fold cross validation and random splitting. We used cross validation since this is the main evaluation method used by the previous studies, i.e., [6], [7], [8], [9]. However, the use of two independent datasets, one for training and one for evaluation is important to get reliable results. Therefore, we randomly split the dataset into two sets of the same size and the same spread between vulnerable and non-vulnerable files. To avoid any bias from the random splitting, we repeated the process 50 times to create 50 distinct experiment settings. The random split forms an attempt to evaluate based on a "fresh" set of data, as required by the machine learning guidelines and differs from the cross validation since it does not systematically learn and evaluate.

To answer RQ2, we used the realistic dataset and random splitting repeated 50 times.

For RQ3, we split our datasets according to 20 reference points. Each reference point corresponds to the release dates of the Linux kernel releases, from the 2.6.28 (released 25 December 2008) to the 3.7 (released 10 December 2012). For every reference point, we train on the selected files retrieved before the reference date and evaluate (test) on the files that were retrieved after it.

2) Effectiveness Measurements: All the approaches replicated used binary classification of their results. Thus, they have 4 outcomes: false positives (FPs), false negatives (FNs), true positives (TPs) and true negatives (TNs). In our case, a FP denotes a file incorrectly classified as vulnerable, while a FN is a file incorrectly classified as non-vulnerable. A TP and a TN is a file that is correctly classified as vulnerable and non-vulnerable, respectively.

To evaluate the effectiveness of the studied classifiers, we rely on the so-called precision and recall metrics. Precision (P)

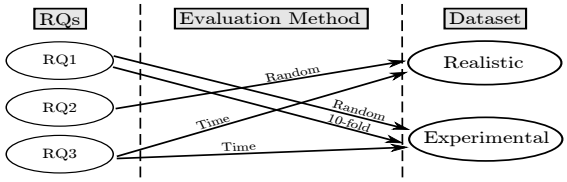


Fig. 1. Evaluation methods and datasets that were used to answer our RQs.

represents the probability that a random file that is classified as vulnerable is indeed a vulnerable one. Recall (R) is the probability that a vulnerable file will be classified as vulnerable by our classifier. They are calculated as follows:

$$P = TP/(TP + FP)$$
$$R = TP/(TP + FN)$$

However, these two metrics are not enough to evaluate the models more generally, especially against random classification. Hence we also compute the Matthews Correlation Coefficient (MCC)[25] that is largely used in machine learning to measure the quality of binary classifiers.

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

This metric returns a coefficient between -1 and +1, a score of 1 indicates a perfect prediction whereas a score of -1 indicates that all data were wrongly predicted. A coefficient of 0 indicates that the classifier performed as well as a model that classify randomly the data.

B. Results

1) Answering RQ1: Experimental Dataset: Table I records the average precision and recall values, when running 10 times a stratified 10 fold cross validation on the experimental dataset. It should be noted that median values are very close to the average ones. From these data we can observe that two methods are performing reasonably well with an MCC close or equal to 0.6: the includes and function calls and the text mining. These two approaches have precision above 70% which is often mentioned as a "practical threshold". Recall values for these two methods are approximately 60%. The model of software metrics performed worse, with relatively low recall and MCC.

Figure 2(a) shows the precision and recall values of all the studied approaches. We present them under the form of a bag plot to visualise the variation between the performed repetitions of the random splits and ease the comparison. Evidently, the text mining approach is performing best for both precision and recall. It even manages to be above the "practical threshold" with respect to precision; with an average value

TABLE I
PRECISION AND RECALL WITH RESPECT TO CROSS VALIDATION, IN THE EXPERIMENTAL DATASET. (RQ1)

	Software Metrics	**Includes**	**Function Calls**	**Text Mining**
Precision	65%	70%	67%	76.5%
Recall	22%	63%	64%	58%
MCC	0.28	0.59	0.58	0.60

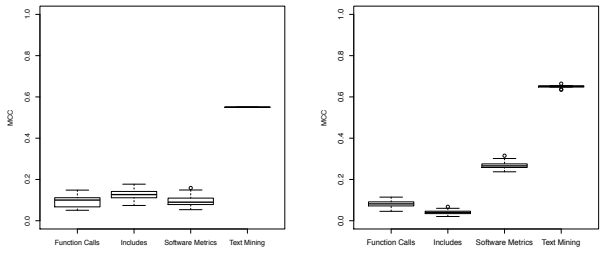
of 76.3%, while its average recall is 53%. This impression is further confirm by the result of the MCC presented in Fig. 3(b) where the text mining approach outperform by far the other approaches. The include & function calls and software metrics while achieving similar MCC score around 0.15, performs differently in regards of precision and recall. The includes and function calls achieves in average 30% of both precision and recall. The software metrics approach provides an interesting precision ranging from 35% to 55% but at the price of a low recall, which is less than 10%.

2) Answering RQ2: Realistic Dataset: RQ2 regards evaluating based on the random split of the realistic dataset. Figure 3(b) displays the MCC results. Evidently, the text mining outperforms the other approach with an average MCC of 0.65. While observing the result of precision and recall (Fig. 2(b)), we can see that this is mostly driven by a great precision close to 100%. The software metrics model is the second best with an average MCC of 0.27 and an average precision of 60%. Include and Function calls are both performing poorly with MCC close to 0.1, meaning that these kinds of approaches in this context perform barely better than random classification.

3) Answering RQ3: Evaluation wrt Time: RQ3 regards the evaluation of the studied methods when using past data to predict future vulnerabilities. Figure 4 depicts the results of the experimental dataset. From these data we can observe that the most precise approach is the text mining. Its best precision, of approximately 78%, is achieved in the first studied release. The includes and function calls are following closely with a precision ranging from 60% on the last release to the maximum 71%, on the 8th studied release. Precision of the

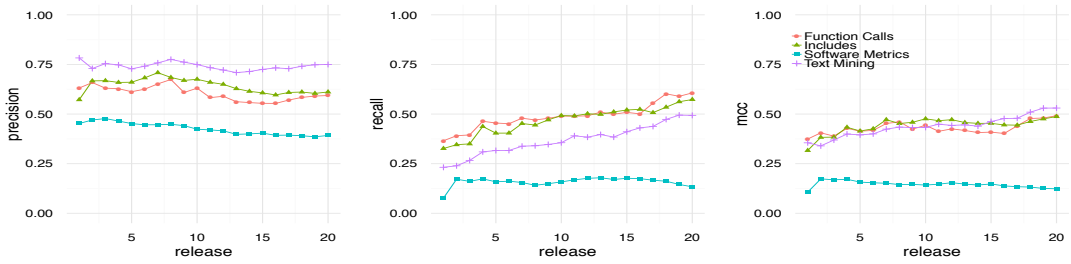
(a) Experimental (RQ1) (b) Realistic (RQ2)

Fig. 2. Bagplot of precision over recall when using random split for the experimental and realistic datasets



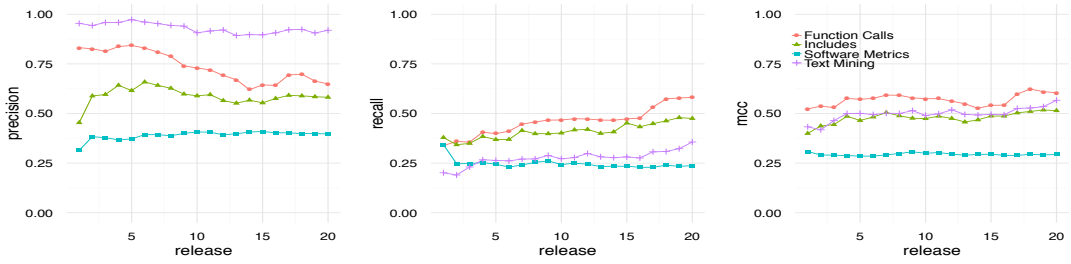
(a) Experimental (RQ1) (b) Realistic (RQ2)

Fig. 3. MCC box plots when using random split for the experimental and realistic datasets.



(a) Precision (b) Recall (c) MCC

Fig. 4. Time split for the experimental dataset (RQ3)



(a) Precision (b) Recall (c) MCC

Fig. 5. Time split for the realistic dataset (RQ3)

software metrics is slowly decreasing from 48% to 40%. All the recall values having an opposing trend, i.e., they improve over time. Surprisingly, the best recall value is obtained by the includes and function calls, with results ranging from 32% to 62%. Text mining is slightly worse, with results from 22% to 50%. Regarding MCC, we observe that Include & Function calls and Text Mining are performing similarly.

Figure 5 presents our results with respect to the realistic dataset. Interestingly, these are not so different from those we got for the experimental dataset. The text mining is still the more precise, but regarding the recall Function calls is performing better managing to get more than half of the vulnerabilities for the latest releases. This can be observed on the MCC graph where Function calls is the best approach for all releases, while Include and Text mining perform similarly.

4) Answering RQ4: Memory and Time Cost: In the previous RQs, we focused on measuring the effectiveness with different settings. It seems that the text mining approach is the one that performs best in most of the cases, but with a price to pay; the time and in memory consumption.

The most important problem here is the cost with respect to the needed memory. Thus, 200 GB of memory was required to run Text Mining, 80 GB and 120 GB for the includes and function calls, respectively, and 4 GB for software metrics. Arguably, these numbers are due to the Weka library, which might not be the best choice for large datasets. Yet, we believe that it provides a good indication of the cost of the studied methods which explodes in the case of text mining.

Regarding the time required to run the different parts of our study, we observed the following: Text mining is extremely fast for the extraction of features but requires quite some time to both train and test up to 3h in the case of the realistic dataset. This is due to the explosion of features caused by the use of bag of words and due to the use of several filters. The software metrics, due to its small number of features, is really fast with respect to training and test (less than 10s). However, the feature extraction phase is really time consuming especially for the git browsing part. Indeed, the Linux kernel has a long history and retrieving the git history of every file is rather time-consuming. The includes and function calls model seems to offer a nice trade-off with a fast time to extract the needed features, a shorter time than text mining to build, learn and evaluate. It is noted that function calls have a higher number of features than includes and require more preprocessing.

VII. Discussion

A. Implications

Our study concerns two scenarios: one that corresponds to the random split of the datasets (referring to the case of that we can distinguish between vulnerable and non-vulnerable files) and one corresponding to the time split, which refers to the more practical case of whether past information is adequate for predicting future vulnerabilities.

A direct implication of our results is that in our context, historical data are good in supporting relatively accurate predictions on future vulnerabilities. This is quite encouraging

TABLE II
COMPARISON WITH RELATED WORK. TIME-BASED RESULTS RECORDED AS (Y / Z) REFER TO (EXPERIMENTAL DATASET / REALISTIC DATASET). THE MARK 'X' DENOTES THE ABSENCE OF REPORTED RESULTS BY THE PREVIOUS STUDY.

		Includes and Function calls		**Software Metrics**				**Text Mining**		
		Neuhaus et al.[7]	This paper	Shin et al.[8]	Shin et al.[13]	Walden et al.[6]	This paper	Scandariato et al.[9]	Walden et al.[6]	This paper
Cross validation	Precision	70	70	3 - 5	9	2-52	65	90*	2-57	76
	Recall	45	64	87 - 90	91	66-79	22	77*	74-81	58
Time	Precision	X	64 / 73	3	X	X	42 / 39	86*	X	74 / 93
	Recall	X	48 / 46	79 - 85	X	X	16 / 24	77*	X	37 / 27

*Estimated from the graphs and reported data of the paper.

since it suggests that vulnerability prediction models can be useful and practical. As shown in Figures 4 and 5 the top performing prediction models achieve precision values of approximately 75% with recall of approximately 50%, which are judged by the studies of Morrison et al. [10] and Shin et al. [13] to be satisfactory. Interestingly we found a small influence of the imbalanced data (as shown by the differences between Figures 4 and 5) on our results. This is in contrast with the results reported by Morrison et al. [10] indicating the need for further research on the impact of data imbalance on the prediction models.

Other important findings highlighted by our results regard the MCC values we found. The MCC coefficient quantifies the quality of the predictions when compared to a random one. Thus, an MCC value equal to 1 represents a perfect prediction, while 0 a random prediction one. Therefore, all of our predictions (under all studied settings) are far better than the random ones (since we get MCC values in the range of 0.25 - 0.6) indicating that the built models do manage to learn relatively well.

Interestingly, the results of the practical case (time split) are closer to those of cross validation (Table I). This can be explained by the fact that Linux kernel vulnerabilities can be well predicted by the historical data (as discussed in the beginning of this section). Thus, most of these data are probably selected by the 9 training folds that are used in every of the 10-fold cross validation iterations. Therefore, we get similar results with. Of course if historical data were not enough, cross validation would probably provide an overestimation of the models' performance.

Overall, our data suggest that in the practical case (time split) the most effective approaches are the one based on the includes and the function calls. This is somehow surprising since this method was proposed long time before the other ones which use much more sophisticated methods. By contrast, the text mining technique is by far the best one when considering the general scenario or favouring precision over recall. This could be seen as an ability of text mining to easily learn what the training data have to offer, whereas include and function calls require a more representative training dataset.

B. Differences with Previous Studies

In this study, we replicated all the existing vulnerability prediction methods. A natural question to ask is how our results compared with the ones reported. Table II summaries this data. Interestingly, there are some differences especially regarding the recall values. Regarding the includes and function calls approach [7], our data are in line with those reported in literature (case of cross validation), i.e., same precision values with slightly better recall ones.

With respect to the software metrics approach, we found different results from all the other studies [8], [13], [6], i.e., better precision and significantly lower recall. This difference could be explained by the way we construct our datasets and/or by the fact that undersampling is used in these studies to balance the datasets which may have impacted the drawn result.

Finally, with respect to the text mining approach we observed comparable result in terms of precision with the study of Scandariato et al. [9]. The recall found was, however, lower than the one claimed, up to 50% in the case of time splits. In the study of Walden et al. [6] there was a large variation on the reported results, i.e, 2% of precision in a case while, 57% on another one. This makes a comparison with our study difficult and probably irrelevant. Nevertheless, the same study only used cross validation where we found a better precision and worse recall than them.

C. Threats to Validity

1) Construct Validity: We automatically created our dataset based on the available data within the git commit messages and the CVE-NVD database. However, this process ensures the retrieval of known and fixed vulnerabilities and thus, undiscovered or non-fixed vulnerabilities might be false negatives with a potential impact on our measurements. However, given the size of the Linux kernel and the long history of vulnerability reports, we believe that it is unlikely to have many such cases.

In a similar manner, we used bugzilla links to retrieve bugs. Yet developers might not report there all bugs with the effect of restricting our data to a subset of bugs. This could impact the result of RQ1 as some bugs might not been considered. However, our aim is to distinguish between vulnerable and other, likely to be buggy, files, which are those we collected. Additionally, in RQ3 we used the whole past and future data so that when time evolves the training set becomes larger, while the test set becomes smaller. While, this is an intuitional choice, it is possible that by constraining the training and evaluation sets to specific time limits might improve our results.

2) Internal Validity: This work only considers source code files written in C, but these are not the only files that can be linked to vulnerabilities. For instance, there are parts of the Linux kernel which are written using assembly code. However, since the great majority of the Linux kernel is written in C, it limits the impact of this threat.

Additionally, potential bugs in our implementation may also influence our results. Also we might unintentionally not re-implement exactly the original approaches. To reduce these threats we carefully inspected all of our code, parameters and experiment decisions with respect to the exact replication of the previous approaches. We also manually tested and verified our implementation. Since our results are in line with the previously published ones, we believe that these threats are not of particular importance.

Furthermore, following the suggestion of Shin et al.[8], we used the three best metrics according to Information Gain to build the software metrics model. Still there is a possibility that additional metrics could provide different results.

3) External Validity: The study is limited to the Linux Kernel and thus, our results might not be generalizable to other projects or contexts. However, we studied a real, large and widely used project; the Linux kernel. Also, we studied a large number, much larger than any previous study, of real vulnerabilities (actually all reported vulnerabilities in NVD). As a result, we are confident that our results are accurate at least for the specific context. Unfortunately, additional studies are required to adequately deal with the generalization threat.

D. Future Work

The present study forms a first step towards building security-related prediction models. It can be extended in the following directions:

- Investigate the generalisation of our results on other security sensitive projects.
- Investigate the creation of a composite model that can combine the strengths of the all the approaches.
- Specialise the prediction models on specific types of vulnerabilites
- Built models that can work and characterise commits.

VIII. Conclusions

Researchers have proposed prediction models that highlight likely vulnerable parts of the systems under analysis. In this paper, we presented a large-scale study that aimed at reliably comparing the main prediction models in the context of the Linux kernel. We showed that several parameters like the evaluation method, application scenario and the composition of the dataset, often ignored in literature, can have a major impact on the reported results. We also demonstrated that when aiming at predicting future vulnerabilities, the includes and function calls are the best performing approaches, while when aiming at random instances of vulnerabilities, the best one is text mining. Good news are that the prediction models show a great precision and an interesting recall, hence could be of great help in the prioritization of code reviews.

Acknowledgement

All experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [26].

References

[1] M. Howard and S. Lipner, *The Security Development Lifecycle*. Redmond, WA, USA: Microsoft Press, 2006.

[2] G. McGraw, "Automated code review tools for security," *IEEE Computer*, vol. 41, no. 12, pp. 108–111, 2008.

[3] I. V. Krsul, "Software vulnerability analysis," Ph.D. dissertation, West Lafayette, IN, USA, 1998.

[4] National vulnerability database:. [Online]. Available: https://nvd.nist.gov

[5] G. McGraw and B. Potter, "Software security testing," *IEEE Security & Privacy*, vol. 2, no. 5, pp. 81–85, 2004.

[6] J. Walden, J. Stuckman, and R. Scandariato, "Predicting Vulnerable Components: Software Metrics vs Text Mining," in *ISSRE'14*, pp. 23–33.

[7] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *CCS'07*, p. 529.

[8] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *IEEE TSE*, vol. 37, no. 6, pp. 772–787, Nov. 2011.

[9] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting Vulnerable Software Components via Text Mining," *IEEE TSE*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.

[10] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *HotSoS'15*, 2015, pp. 4:1–4:9.

[11] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Software Eng.*, vol. 38, no. 6, pp. 1276–1304, 2012.

[12] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *QoP'08*, p. 31.

[13] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, Feb. 2013.

[14] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *ICST'10*, pp. 421–428.

[15] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *SAC'10*, p. 1963.

[16] ——, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294 – 313, 2011, special Issue on Security and Dependability Assurance of Software Architectures.

[17] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *MetriSec'10*, pp. 3:1–3:8.

[18] H. Hata, O. Mizuno, and T. Kikuno, "Fault-prone module detection using large-scale text features based on spam filtering," *Empirical Softw. Engg.*, vol. 15, no. 2, pp. 147–165, Apr. 2010.

[19] P. M. Domingos, "A few useful things to know about machine learning," *Commun. ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[20] M. Jimenez, M. Papadakis, T. F. Bissyande, and J. Klein, "Profiling android vulnerabilities," in *QRS'16*. IEEE, 2016.

[21] B. Smith and L. Williams, "Using SQL hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities," in *ICST'11*.

[22] M. Gegick, P. Rotella, and L. Williams, "Predicting Attack-prone Components," in *ICST'09*. IEEE, pp. 181–190.

[23] ——, *ESSoS'09*, ch. Toward Non-security Failures as a Predictor of Security Faults and Failures, pp. 135–149.

[24] I. Kononenko, "On biases in estimating multi-valued attributes," in *IJCAI'95*, pp. 1034–1040.

[25] B. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA) - Protein Structure*, vol. 405, no. 2, pp. 442 – 451, 1975.

[26] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic hpc cluster: The ul experience," in *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*. Bologna, Italy: IEEE, July 2014, pp. 959–967.