An Exploratory Study on Machine Learning to Combine Security Vulnerability Alerts from Static Analysis Tools

José D'Abruzzo Pereira, João R. Campos, Marco Vieira
Department of Informatics Engineering
University of Coimbra
Coimbra, Portugal
josep@dei.uc.pt, jrcampos@dei.uc.pt, mvieira@dei.uc.pt

***Abstract*—Due to time-to-market needs and cost of manual validation techniques, software systems are often deployed with vulnerabilities that may be exploited to gain illegitimate access/control, ultimately resulting in non-negligible consequences. Static Analysis Tools (SATs) are widely used for vulnerability detection, where the source code is analyzed without executing it. However, the performance of SATs varies considerably and a high detection rate usually comes with significant false alarms. Recent studies considered combining various SATs to improve the overall detection ability, but they do not allow exploring different performance trade-offs, as basic and rigid rules are normally followed. Machine Learning (ML) algorithms have shown promising results in several complex problems, due to their ability to fit specific needs. This paper presents an exploratory study on the combination of the output of SATs through ML algorithms to improve vulnerability detection while trying to reduce false alarms. The dataset consists of SQL Injection (SQLi) and Cross-Site Scripting (XSS) vulnerabilities detected by five different SATs in a large set of WordPress plugins developed in PHP. Results show that, for the case of SQLi, a false alarm reduction is possible without compromising the vulnerabilities detected, and that using ML allows trade-offs (e.g., reduction in false alarms at the expense of a few vulnerabilities) that are not possible with existing techniques. The paper also proposes a regression-based approach for ranking source code files considering estimates of vulnerabilities computed using the output of SATs. Results show that the approach allows creating a ranking of the source code files that largely overlaps the real ranking (based on real known vulnerabilities).**

***Index Terms*—Security, Vulnerability Detection, Static Code Analysis, Machine Learning**

I. Introduction

Security risks, business needs, and industry changes are listed as the top three drivers for security spending according to a survey conducted by Gartner in 2017 [1]. At the same time, the implementation of the General Data Protection Regulation (GDPR) and other security and privacy regulations are driving change and growth in the security services market [2]. For example, among many other aspects, such regulations require companies to report personal data breaches, otherwise they can get fines that may compromise their revenue. The problem is that software systems are often deployed with vulnerabilities that can open the door for successful attacks.

A software vulnerability is a weakness in a system, and it represents a security risk. When exploited, it may lead to an intrusion [3] with severe consequences, including financial and data losses. There are many techniques to prevent software vulnerabilities, including best coding practices and vulnerability detection approaches and tools [4]. For example, security requirements can be elicited and prioritized using the Security Quality Requirements Engineering (SQUARE) methodology [5]; during the application development phase, software developers can use guidelines such as the Open Web Application Security Project (OWASP) secure coding practices to avoid introducing vulnerabilities [6]; and when the code source is ready, either static (such as Static Code Analysis (SCA)) or dynamic (such as penetration testing) approaches can be used.

SCA is one of the most used design-time techniques to detect vulnerabilities, especially through Static Analysis Tools (SATs). Many different automated SATs are available in the field (including open-source and commercial ones). However, although practice shows that SATs are typically able to achieve a high vulnerability detection coverage, they usually report a high number of false alarms [7], also known as False Positives (FPs). This happens because the SATs have to make approximations depending on the detection approach being used (such as finding patterns in the source code) and on the size and complexity of the code being analyzed. As the penalty for not detecting a vulnerability may be high, the preference is to not lose vulnerabilities, which may naturally lead to an increase in the number of FPs [8].

An obvious approach to reduce the number of False Negatives (FNs) (i.e., unreported vulnerabilities) is to combine the output of diverse SATs. Combination heuristics that suggest raising an alert when one of a set of SATs reports an issue were already addressed in the literature [9]. However, this leads to a significant increase in the FPs as it multiplies the potential sources for vulnerability detection mistakes. Hence, alternative techniques are needed to reduce false alerts.

Machine Learning (ML) techniques have been used in several areas, including software engineering. Their application reveals good results for classification problems and

978-1-7281-6622-3/19/$31.00 ©2019 IEEE

some studies have already used ML to detect vulnerabilities [10] [11] [12]. However, to the best of our knowledge, no study has explored the use of SCA outputs as input for the ML classifiers.

The contribution of this paper is twofold:

- **explore the use of ML classification algorithms to detect software vulnerabilities using the output of different SATs**, aiming at reducing the number of FPs without compromising the ability to detect vulnerabilities;
- **validate the possibility of creating ranked lists of vulnerable source code files using the output of SATs** to guide the analysis by software development teams. This is particularly important in projects with a high number of files, as well as with time and budget constraints, which is the reality of most projects.

To support our study we use a dataset of vulnerabilities detected by five SATs on a large set of WordPress plugins developed in PHP [13]. We focus the study on two of the most critical vulnerability types for web application security, SQL Injection (SQLi) and Cross-Site Scripting (XSS) [14]. Comparing with the traditional 1-out-of-N (1ooN) heuristic (where an alert is raised when 1 of N detectors raises an alarm [9]), results show an improvement for the case of SQLi vulnerabilities (namely a reduction in the false alarms). On the other hand, for XSS vulnerabilities the results using ML are equal to the ones obtained using 1ooN. Additionally, results show that it is possible to create a ranking of the source code files considering an estimation of the potential number of vulnerabilities in each file, as reported by the multiple SATs (a clear improvement when compared to the use of the alerts raised by any SAT individually). In practice, this information can be used by the developers to prioritize their work and focus on the files with more vulnerabilities.

The rest of this paper is structured as follows. Section II presents background and related work. The exploratory study to predict vulnerabilities from the SATs output is presented in Section III. The approach for ranking vulnerable files is presented in Section IV. Section V discusses threats to validity, and Section VI concludes the paper and puts forward ideas for future work.

II. Background and Related Work

This section presents background concepts and related work on vulnerability detection using SCA and ML algorithms.

A. Vulnerability Detection and Static Code Analysis

There are several techniques to detect software vulnerabilities. A survey by *Liu et al.* [4] details four of the most used ones: SCA, fuzzing, Software Penetration Testing (SPT), and Vulnerability Discovery Model (VDM). *SCA* is the "evaluation of a system or component without the program execution" [7]; *fuzzing* is a randomized testing technique that generates random character streams for the tests; *SPT* simulates attacks of malicious users; and *VDMs* are based on software reliability models, and "specify the general form of the dependence of the vulnerability discovery process on the principal factors that affect it". The present study focuses on static code analysis as it does not require the execution of the software to detect vulnerabilities, and is typically able to achieve a high detection rate.

SATs are used to decrease the time needed for security code reviews, which may involve many people and thus be very expensive [15]. Such tools are able to cover the entire source code and can thus be used early in the Software Development Life Cycle (SDLC). The outcome of the SATs are alerts, reporting issues of diverse types, such as memory errors, resource leaks, violation of APIs or framework rules, exceptions, encapsulation violations, race conditions, and security vulnerabilities.

Different SCA techniques can be used to identify potential issues in the code [16]. The simplest one is to *scan the source code for simple patterns* [8]: if a piece of code matches a rule that indicates a problem, then an alert is raised. *Data flow analysis* is a technique [17] where the SAT uses the possible values that variables can have to evaluate if an exception (e.g., a null pointer exception) can happen at runtime. A Java example can be seen below where the third line will raise an exception in case the variable `g` is `null`.

```
if (g != null)
  paintScrollBars(g, colors);
g.dispose();
```

A technique used for detecting vulnerabilities is *taint propagation*. It is a data flow technique that involves tracking tainted sources, such as user inputs, and validating if specific computations are affected by them [7]. SATs use *taint propagation* to detect SQLi and XSS vulnerabilities. When a SQLi vulnerability is exploited, a SQL statement is altered and an attacker can read or modify the database content. An example of a SQL construct that contains a SQLi vulnerability can be seen below:

```
<?php
$email = $_GET['email'];
$password = $_GET['password'];

$mysqli = new mysqli('localhost', 'dbuser',
    'dbpasswd', 'sql_injection_example');
$sql = "SELECT * FROM users WHERE " .
    "username=$email AND password=$password";

if ($result = $mysqli->query($sql)) { /* code */ }
?>
```

For example, if the malicious string `' OR 1=1 --` is used by the attacker instead of an actual email, the expression will be evaluated as true due to the `1 = 1` comparison (tautology). As a consequence, the attacker may gain access to the system if such construct is used to support authentication. To avoid this vulnerability, the inputs should be validated, parameterized queries should be used, and the user data should be sanitized.

A *XSS* vulnerability allows attackers to inject malicious client-side scripts into web pages that will be viewed by other users. Such vulnerabilities are due to the lack of proper validation or escaping of the user-supplied data, and are found

in around two-thirds of all web applications [14]. When this vulnerability is successfully exploited, the attacker is able to execute scripts remotely in the user web browser.

SQLi and XSS attacks target vulnerable Sensitive Sinks (SSs) [18]. A SS is a command that uses external data (e.g., data provided by the user), also called an entry point. For instance, a SQL query that contains entry point(s) is a SS. If one of those entry points is unsafe, it can lead to a SQLi vulnerability. The same happens for XSS vulnerabilities. For instance, the PHP command `echo ''$name $city''` has two entry points that may lead to two distinct XSS vulnerabilities [19].

Several studies compared tools to detect vulnerabilities. For example, *Antunes et. al.* compared the effectiveness of four penetration testing tools with three SATs for detecting SQLi vulnerabilities [20]. The tests were performed on top of vulnerable and non-vulnerable web services, and results show that SATs are able to achieve a higher coverage of vulnerabilities. Nevertheless, both SATs and penetration testing tools reported a considerable number of FPs. Other studies on the topic include the comparison of a SAT with a penetration testing tool [21], and the comparison of different vulnerability detection techniques [22].

Algaith et. al. combined the output of different SATs to decrease the FNs [9]. The results were obtained from the alerts raised by five SATs on a large set of WordPress plugins developed in PHP. Three combination approaches to identify SQLi and XSS vulnerabilities were used: *1) 1-out-of-N*: raises the alert when any of the tools report the alert; *2) N-out-of-N*: raises the alert when all the tools report the alert; and *3) simple majority*: raises the alert when the simple majority of the tools reports the alert. Results show that *N-out-of-N* has a better specificity than *1-out-of-N* or *simple majority*, at the cost of a low recall. The heuristic with the best performance regarding the detected vulnerabilities is *1-out-of-N*, resulting in a very high recall. On the other hand, the number of FPs increases, which reduces the precision. The main limitation of this approach is the inability to explore different performance trade-offs, as basic and rigid combination rules are followed.

B. Machine Learning for Vulnerability Detection

In recent years, ML algorithms have been successfully used in a variety of complex problems, including vulnerability detection. Such algorithms can find complex patterns in the data and learn from them without relying on a predetermined model. Afterwards, they can be used to make predictions on unseen data based on what was learned.

Considering the ML terminology, a Sensitive Sink (SS) represents a sample in the dataset and contains a label. In this case, two values are possible: actual vulnerability (positive) or not (negative). The output of the SATs for each SS are considered as features, and they are part of the sample. Thus, the vulnerability detection problem can be considered as a classification problem, which is concerned with separating data into distinct classes. Examples of classification algorithms include Decision Tree (DT) and Support Vector Machine (SVM) [23].

As the complexity of any model depends on the number of inputs, reducing dimensionality (e.g., *Feature Selection/Extraction*) may be important. Additionally, to help algorithms to cope with the infeasibility of very large datasets, *Instance Selection* techniques (e.g., *sampling*, *boosting* [23]) can be used. To deal with imbalanced datasets, which may compromise the performance of the algorithms on the minority classes, there are specific solutions such as *Undersampling* and *Oversampling*.

To get a realistic estimate of the performance of a model, two main sets of data are normally used: *train* and *test*. The training set is used for training the model, while the testing set is used to estimate its generalization error. This division is not trivial, as it may inadvertently influence the performance/representativeness of the model. Thus, several techniques have been proposed over the years (e.g., *Partition/Leave-one-out*, *Bootstrap Methods* [24]).

Various studies have used ML to predict software vulnerabilities. An example is the study by *Walden et al.* [11], where software metrics (such as the metrics CK [25]) and text mining were used as input for the Random Forest (RF) ML algorithm. Datasets with software metrics and text mining data were created using three PHP Applications (Drupal, Moodle, PHPMyAdmin). Authors analyzed both recall and inspection ratio, and results show that a higher recall is obtained when using text mining data than when considering software metrics.

Alves et. al. used several ML classifiers (*Naïve Bayes*, *Decision Trees*, *Random Forest*, and *Logistic Regression*) to predict vulnerabilities in `C/C++` projects using datasets of software metrics [12]. A dataset contains metrics obtained both before and after applying the patches for a number of vulnerabilities previously reported in the Common Vulnerability and Exposures (CVE) repository. The following projects were considered: *Mozilla*, *httpd*, *glibc*, *Linux Kernel*, and *Xen Hypervisor*. In general, results showed a low precision (from 0.32% to 30.50%) and a wide range for recall (from 0.36% to 100.0%).

III. Exploratory Study of ML Techniques

This section presents the exploratory study on the use of ML classification algorithms to detect software vulnerabilities using the output of different SATs. The goal is to reduce the number of FPs without compromising the ability to detect vulnerabilities. We present the dataset used, introduce the ML algorithms tested, and discuss the results.

Fig. 1 depicts the overall process. As shown, the SATs are run in the same source code. Then, the alerts are combined in a dataset, where each sample refers to a SS and contains the output of each SAT (alert raised or not). The ML algorithms are run using the dataset, and a prediction (either as vulnerable or non-vulnerable) is given to each sample in the dataset. The performance of each ML model is analyzed considering precision, recall and F-Measure metrics.

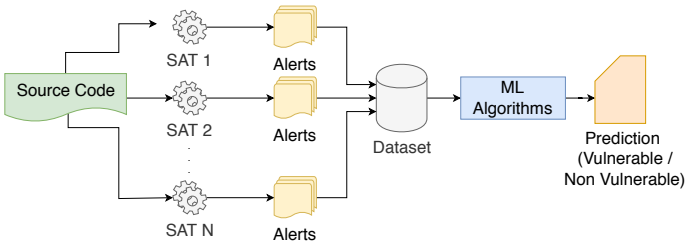

Fig. 1. Overall methodology to combine diverse SATs.

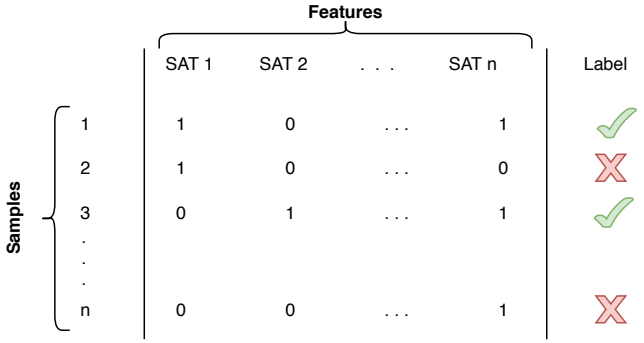

Fig. 2. Representation of the dataset with samples and features

A. Datasets

Two datasets were considered in this study, one with known SQLi vulnerabilities and the other with XSS vulnerabilities [26]. In practice, the two datasets are based on the alerts raised by five different SATs that were used to analyze 134 WordPress plugins developed in PHP. WordPress is a widely used Content Management System (CMS) and many websites of all sizes are built with it. Also, it is based on one of the most used programming languages for the web: PHP [27]. The SATs used to detect vulnerabilities are: *phpSAFE* [28], *RIPS* [29], *WAP* [30], *Pixy* [31], *WeVerca* [32]. They are well-known SATs for PHP, and quite referenced in the literature as security review tools. The datasets have been created by *Nunes et al.* and first used in [13]. In our study, we considered their updated version from [19].

Each sample (line) in the datasets corresponds to a SQLi or XSS sensitive sink and contains the following information: *a)* file name of the SS, *b)* line number of the SS, *c)* report of the five SATs (either 1 when an alert was reported by the SAT or 0 otherwise), and *d)* label that indicates if the SS contains real a vulnerability or not. A representation of the datasets can be seen in Fig. 2. The original datasets (from *Nunes et al.* [19]) include information for all SSs in the plugins, being they vulnerable or not.

The WordPress plugins have 466,164 Logical Lines of Code (LLOC), and the number of SSs is 5,216 for SQLi, and 25,290 for XSS. From the datasets provided by *Nunes et al.*, we removed the vulnerable SSs that were not reported by any SAT, as they could lead the ML algorithms to wrong predictions. In practice, these SSs were originally identified as vulnerable not because they were reported by static analysis but after either a manual review process or through the report of the vulnerability on the WPScan Vulnerability Database (WPVD) [26] (which makes them out of the scope of this study). The non-vulnerable SSs that were not reported by any tool were also removed, as they do not add any relevant information to the ML models and their elimination allows reducing the time/model complexity. Hence, the filtered datasets contain only the samples (SSs) that have at least one alert raised by one of the five SATs. Additionally, the alerts regarding plugins that are not available in the WordPress repository anymore were also removed (20 for SQLi and 9 for XSS).

As the samples (sensitive sinks) in the datasets are already classified with two labels (vulnerable/non-vulnerable), we can classify each alert reported by each SAT according to the following: *a)* **True Negative (TN)**: a SS not reported by a SAT as a vulnerability and that is not an actual vulnerability (*not vulnerable code*); *b)* **False Positive (FP)**: a SS reported by a SAT as a vulnerability that is not an actual vulnerability (*not vulnerable code*); *c)* **False Negative (FN)**: a SS not reported by a SAT as a vulnerability that is an actual vulnerability (*vulnerable code*); and *d)* **True Positive (TP)**: a SS reported by a SAT as a vulnerability that is an actual vulnerability (*vulnerable code*).

The datasets that resulted from the filtering discussed above contain 928 SQLi samples (670 (72.20%) vulnerable SSs and 258 (27.80%) non-vulnerable SSs), and 5,968 XSS samples (4,930 (82.61%) vulnerable SSs and 1,038 (17.39%) non-vulnerable SSs). The datasets are thus unbalanced, as there are many more samples in the positive class (vulnerable SSs) than in the negative one (non-vulnerable SSs). Hence, sampling techniques should be considered before running the ML algorithms. Normalization of the features is not needed as they are all in the same scale: they either represent the presence of an alert in a given SS (value 1) or not (value 0).

It is important to mention that, although the SATs report alerts for a particular type of vulnerability, it can not be said that each True Positive (TP) corresponds to a single vulnerability. In fact, an alert may correspond to more than one vulnerability if the SS uses more than one entry point (as in the SQL example in Section II). The results presented in this paper refer to the vulnerable SSs and not the total number of vulnerabilities, as the SATs report one occurrence for each vulnerable SSs.

B. Machine Learning Algorithms and Techniques

Several ML algorithms were initially experimented using the Propheticus tool [33], and the ones that showed more promising results were studied in more detail considering a large list of configurations, as described in this section.

The list of algorithms used in the study, as well as the *feature selection* and *sampling* techniques applied are listed in *Table I* (details on each technique can be found in [23]). Although not all the possible parameter configurations are listed in the table (due to space constraints), they can be easily obtained by combining the different values: for example, applying *variance* and *correlation* at the same time for feature

TABLE I
ALGORITHMS AND TECHNIQUES

Parameter	Values
Feature Selection	Variance, Correlation
Sampling	Random under/oversampling, Synthetic Minority Over-Sampling Technique (SMOTE)
Algorithms	DT, Neural Network (NN), RF, SVM, Gradient Boosting (GB), Bagging

TABLE II
ALGORITHMS' HYPERPARAMETERS

Alg.	Hyperparameters
DT	**criter.**: [gini, entropy], **min_samp_split**: [.001, 2], **max_feat.**: [.1, .55, 1.0], **min_samp_leaf**: [.001, 1]
NN	**hidden_layers**: [(100,1), (100, 20), (100, 20, 10)], **activation**: [logistic, relu], **solver**: [sgd, adam], **learning_rate**: [constant, invscaling, adpative]
RF	**estimators**: [10, 50, 100, 200], **max_feat.**: [.1, .55, 1.0], **criter.**: [gini], **min_samp_leaf**: [.001, 1], **min_samp_split**: [.001, 2], **bootstrap**: [1, 0]
SVM	**kernel**: [linear, polynomial], **C**: [.01, .1, 1], **gamma**: [.1, 1], **degree**: [2]
GB	**estimators**: [50, 100, 200], **learning rate**: [1, .1], **min_samp_leaf**: [.001, 1], **max_feat.**: [.1, .55, 1.0], **min_samp_split**: [.001, 2]
Bagging	**max_features**: [.1, .55, 1.0], **bootstrap**: [1, 0], **estimators**: [50, 100, 200]

selection, or *random undersampling* and *Synthetic Minority Over-Sampling Technique (SMOTE)* for sampling. The values used for the hyperparameters of each algorithm can be seen in *Table II*. Their values were also combined and submitted to the Propheticus tool. For example, for the DT algorithm we have 24 configurations (as three of the hyperparameters have 2 possible values and one has 3 possible values: $2^3 * 3 = 24$).

To evaluate the results, the datasets were divided into training and testing sets. The *training set* was used to train the model, while the *testing set* was used to evaluate it. The method to split the datasets into training and testing is Cross Validation (CV). It consists of dividing the datasets in *k* folds, and use $k-1$ for training and the remaining one for testing. Then, one of the folds used for training is replaced by one used for testing, and the model is trained and tested once again. After *k* iterations, the whole dataset has been used for training and testing [23]. This process aims at avoiding problems like overfitting or selection bias. Note that each of the folds should be divided in a manner that the proportion of the classes remains approximately the same as in the original dataset. This process is called *stratification* [23]. In our study, stratification aimed at maintaining not only the proportion of the classes in each the folds (5-folds were used) but also the proportion of features (SATs in this case). For instance, if a set of SSs is reported as vulnerable by only one SATs, we make sure that these SSs are evenly distributed across the folds.

To add variability to the results, the fold partitioning was repeated 5 times per configuration tested. Each partition is called a seed, and this is used to avoid that a particular configuration presents the best results by chance. Consequently, the results reported by the Propheticus tool are the values multiplied by the seed number (5 in this case). For example, if the number of FPs is 50, the reported number is $50 * seed\_number = 50 * 5 = 250$. It is important to notice that this does not affect the conclusions of the study, as the performance metrics are presented as proportions.

Precision, *Recall*, and *F-Measure* are the metrics used to characterize the performance of each model. **Precision** represents the ratio of TP among all the identified values, and it can be calculated using the equation $precision = TP/(TP+FP)$. **Recall** represents the ratio of identifying the TP among all the positive values, and it can be calculated using the equation $recall = TP/(TP + FN)$. **F-Measure** is the harmonic mean of *precision* and *recall*, and it can be calculated using the equation $F - Measure = 2 * \frac{precision*recall}{precision+recall}$.

It is important to emphasize that, when computing the performance metrics for each model we take into account all the SSs, including the ones for which no SAT alerts were generated (and that were removed from the datasets to reduce time/model complexity, as explained in Section III-A). This allows comparing the results of our study with other approaches, namely with the results reported by *Algaith et. al.* [9] for the original datasets from *Nunes et al.* [19].

C. Results and Discussion

This section presents the results obtained. We start by presenting the results using the 1ooN diversity heuristic. Then, we present an analysis of the dataset using Venn diagrams. Finally, we present the results of the ML algorithms and compare them with the ones obtained with the 1ooN heuristic.

1) Combination of SATs using the 1ooN Heuristic: Algaith et. al. [9] proposed a set of state-of-the-art heuristics to combine the output of SATs for vulnerability detection: *a) 1ooN*: raises an alert when any of the SATs raises the alert; *b) N-out-of-N (NooN)*: raises an alert when all the SATs raises the alert; and *c) Simple Majority Voting*: raises an alert when the simple majority of the SATs raises the alert. As baseline for our exploratory study, we consider the *1ooN* heuristic, with the number (N) of SATs equal to 5. According to *Algaith et. al.* [9], this heuristic has the best performance when considering *recall*, as the number of FNs is the lowest one. Both *simple majority* and *NooN* have a higher precision, but that comes at the cost of a significant decrease in the number of detected vulnerabilities.

The detailed *1ooN* results are presented in *Table* III, and the respective performance metrics (precision, recall, and F-measure) are presented in *Table* IV. Note that, the numbers slightly differ from the *Algaith et al.* paper [9] as we removed some out-of-scope samples from the dataset (as discussed in

Section III-A). As it can be noticed, the recall for both vulnerability types is high (0.931 for SQLi and 0.999 for XSS). This is related to the fact that almost all known vulnerabilities are identified by at least one of the SATs. Additionally, a good precision is obtained for both vulnerability types (0.722 for SQLi and 0.826 for XSS). As the number of FPs is higher than the number of True Negatives (TNs), the precision values are not as good as the recall values. This is due to the higher number of FPs raised by the use of this heuristic (that joins together the FPs of all the SATs). Although the performance metrics are already quite satisfactory, decreasing the number of the FPs is obviously the most promising approach for improving the overall results (*Precision* and *F-Measure*).

TABLE III
RESULTS USING THE 1OON DIVERSITY HEURISTIC FOR XSS AND SQLI VULNERABILITIES

	TN	FP	FN	TP
SQLi	94.66% (4,570)	5.34% (258)	6.94% (50)	93.06% (670)
XSS	95.18% (20,505)	4.82% (1,038)	0.10% (5)	99.90% (4,930)

TABLE IV
METRICS FOR THE 1OON DIVERSITY HEURISTIC FOR XSS AND SQLI VULNERABILITIES

	Precision	Recall	F-Measure
SQLi	0.722	0.931	0.813
XSS	0.826	0.999	0.904

2) Analysis of the SATs Outputs: As the metrics obtained when using the *1ooN* heuristic are already very good, it is important to analyze the relationship among the alerts raised by the several SATs. To this end, we generated Venn diagrams for both TP and FP, which allow understanding the number of vulnerabilities that are detected exclusively by one SAT or by multiple SATs. Fig. 3 and Fig. 4 show the Venn diagrams for SQLi and XSS vulnerabilities, respectively.

When analyzing the SQLi Venn diagrams, it can be noticed that only one true vulnerability is reported by all the SATs. Additionally, *phpSAFE* is the SAT that identified the largest number of TPs, but it is also the responsible for most of the FPs. Moreover, *Pixy* does not detect any vulnerabilities that are not reported by the remaining SATs. Overall, we can see that the different SATs complement well each other with respect to TPs. On the other hand, when it comes to FPs, a big overlap can be noticed. In fact, all the FPs reported by *WAP*, *Pixy*, and *WeVerca* are also reported by at least another SAT (this is why their region is equal to zero in the diagram). This means that the FPs are shared by at least two SATs. Another observation is that no TP is reported in the intersection of *RIPS-WeVerca*, but a high number of FPs are observed (23).

Regarding XSS, a higher number of vulnerabilities (54) are detected by all SATs, when compared to SQLi. *RIPS* is the SAT with the largest number of exclusive alerts for both TPs (1,470) and FPs (490). As noticed for SQLi, *Pixy* does not report any TP that is not reported by any of the remaining SATs. Additionally, neither *Pixy* nor *WeVerca* report any FP exclusively.

The likelihood of an alert being a FP is smaller when more SATs report it. The diagrams show that no FP is reported by all SATs for both SQLi and XSS vulnerabilities. For SQLi, the 4-SAT intersection also does not contain any FP, and almost all the intersection groups of three SATs do not contain FPs (except for the *RIPS-Pixy-WeVerca* intersection). Although the number of FPs for SQLi is small in the 4-SAT and 3-SAT intersection groups, the same conclusion cannot be made for the XSS vulnerabilities. This may be related to the size of the datasets: as there are more XSS than SQLi samples, the likelihood of the group intersections (alerts reported by the several tools) having items is higher (the number of groups is the same). Another possible explanation is related with the nature of the problem: SATs are usually more prepared for SQLi than for XSSs vulnerabilities.

3) Using Machine Learning: As presented before, the baseline results using the 1ooN heuristic are already very good, as most of the true vulnerabilities are detected with a reasonable rate of FPs. Still, the main possibility for improvement is on the reduction of the FPs, a challenging problem to be addressed by applying ML techniques. Note that this approach also allows considering trade-offs (such as reducing the number of FPs at the cost of missing some TPs) that are not possible using the *1ooN* heuristic.

The results obtained are represented in the form of a Confusion Matrix (CM), which is a graphical representation of both the actual value (rows) and the predicted value (columns). Figures 5 and 6 present two examples that we will discuss later in the paper. As shown, the negative values (non-vulnerable - TNs and FPs) are presented in the first row, while the positive values (vulnerable - FNs and TPs) are presented in the second row. The cells contain both the percentage and the count of samples. For space reasons, in the next paragraphs we will discuss only the results of the best ML algorithm configurations for both SQLi and XSS vulnerabilities (detailed results can be found online[1]).

As mentioned in section III-B, the results are multiplied by 5 (due to the seed number used in the study). Also, even though we have removed from the dataset the non-vulnerable SSs and the vulnerable SSs without SAT alerts (to reduce complexity and because vulnerable SSs without SAT alerts are not useful in the context of this study), the results reported consider all the existing SSs. This allows creating the CMs and also comparing them with the results obtained with the *1ooN* heuristic.

Fig. 5 presents the CM for the best results obtained for SQLi vulnerabilities. In this case, the *Bagging* algorithm is used, with no sampling technique. The performance metrics for this case are: precision of 0.737, recall of 0.931, and F-measure of 0.822. These results are slightly better than the *1ooN* baseline (an increase in the precision, from 0.722 to 0.737, while maintaining exactly the same recall), which is due to the decrease of 18.4 FPs (that corresponds to 7.08% less false alarms than in the *1ooN* baseline - decimal values related

[1]Detailed results can be found at https://eden.dei.uc.pt/~josep/LADC2019/

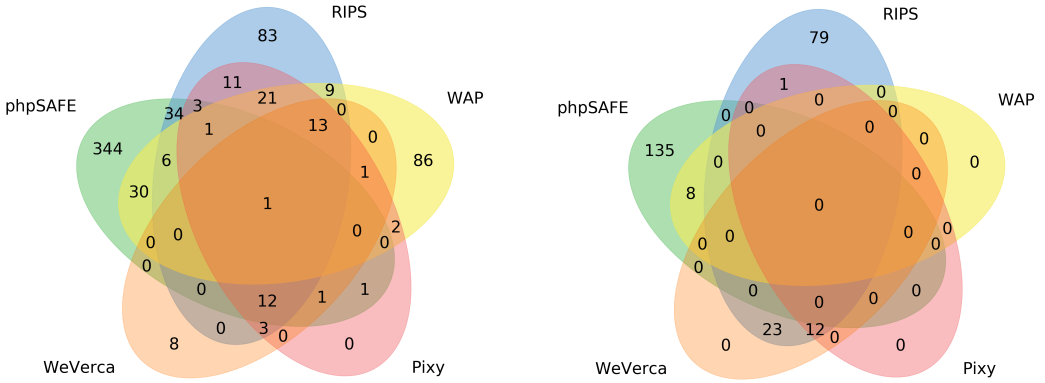

Fig. 3. Venn Diagrams of True Positives (left) and False Positives (right) for SQLi vulnerabilities

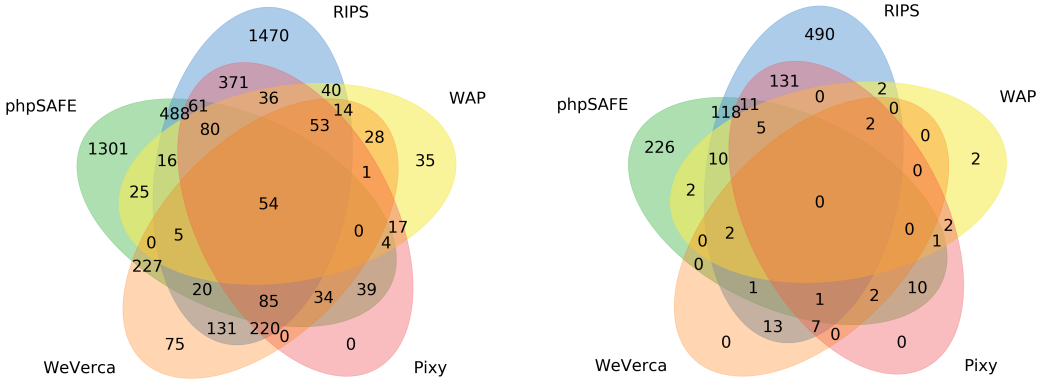

Fig. 4. Venn Diagrams of True Positives (left) and False Positives (right) for XSS vulnerabilities

to the fact that we average the FPs according to the number of seeds/executions). Consequently, the F-measure also increases (from 0.813 to 0.822).

Venn diagrams based on the predictions were also created to understand and interpret the improvements (not included here due to space constraints, but available online (see footnote 1). Comparing the intersection regions of the different diagrams (Fig. 3 and the Venn based on the predictions) allows understanding in which intersection the improvements are noticed. For example, we observed a reduction in the FPs, from 23 to 4.6, in the *RIPS-WeVerca* intersection/combination. Analyzing the Venn diagram for TPs, we see that no TP is detected exclusively by *RIPS* and *WeVerca*. In general, we can conclude that the *Bagging* algorithm is able to learn this pattern, and that is why it is able to provide better results than the *1ooN* heuristic.

Fig. 6 presents the CM of the best configuration for XSS. This configuration uses the *RF* algorithm, with no sampling technique. The performance metrics in this case are: precision of 0.826, recall of 0.999, and F-measure of 0.904. Comparing with the *1ooN* heuristic, we can see that the results are the same (this is also confirmed by the analysis of the different Venn diagrams). This happens because the ML algorithm classified all the SSs having any SAT alert as vulnerable. Consequently, no changes in the number of FPs or FNs are

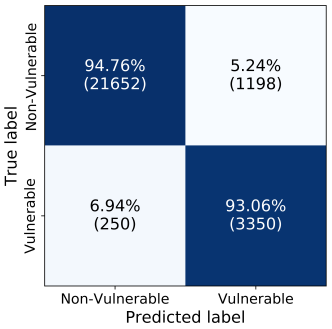

Fig. 5. CM for SQLi vulnerabilities of the prediction (*Bagging*)

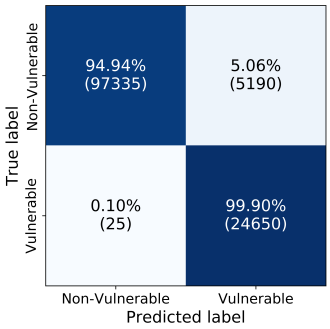

Fig. 6. CM for XSS vulnerabilities of the prediction (*RF*)

noticed, leading to the same overall performance metrics. Nevertheless, as the baseline *1ooN* results are already very good, we can conclude that good performance can also be achieved with ML algorithms.

In an attempt to improve the results, we removed *Pixy* from the dataset, as it does not report alerts that are not reported by the other SATs (the *Pixy* region in the Venn diagrams does not have any items/alerts, as shown in Fig. 4). However, the results obtained neither improved (confirming the vulnerable SSs or rejecting possible FPs) nor deteriorated, which means that no further conclusion can be made.

Although the results obtained with ML are not significantly better than the ones with the *1ooN* heuristic, for the case of SQLi vulnerabilities, there are some configurations that allow reducing the FPs at the cost of missing some TPs. This is a trade-off that cannot be explored using the *1ooN* approach, and is very useful when the time and budget are not enough for analyzing all the potential vulnerabilities. For instance, a reduction of 24.8 FPs (9.61% reduction) can be achieved in some cases at the cost of missing 2.4 TPs (0.36% reduction). In other cases, a reduction of 35.0 FPs (13.57%) can be achieved at the cost of 3.0 TPs (0.45%). Trade-offs can also be considered for XSS vulnerabilities, although our results show a higher penalty in the TPs. For instance, a reduction of 11.2 FPs can be obtained at the cost of 31.0 TPs.

IV. Prioritizing Vulnerable Files

As most of the projects have time and resource constraints, developers need approaches that help them prioritizing their work (in addition to having fewer FPs when analyzing SAT alerts). This way, devising a mechanism to rank the source code files in terms of the potential vulnerabilities that have to be analyzed/fixed can help development teams to focus on the ones with the biggest potential to be vulnerable. In this section we study the possibility of creating ranked lists of vulnerable source code files using the output of SATs. The approach is based on regression algorithms, where the variables are the output of the SATs.

A. Dataset

The same set of WordPress plugins is used to create the dataset for this study. As the PHP files may have more than one vulnerability, which may result in more than one alert reported per file, we need to compute the total number of vulnerabilities per file.

The representation of the dataset can be seen in Fig. 7. The samples are the files, and the process to create the dataset consists of two steps: *1)* count the number of true vulnerabilities per file for each vulnerability types; and *2)* count the number of alerts reported by each SAT. In practice, the dataset includes the following features: *a)* file name; *b)* number of alerts reported by each SAT; *c)* binary value (0 or 1) that, for each SAT, indicates if the SAT reported alerts in the file; and *d)* the number of true vulnerabilities per file (equivalent to the label in the previous section). The features *b* and *c* repeat for the five SATs.

B. ML Techniques

Differently from the ML algorithms studied in the previous section, whose prediction is a categorical value in two classes (vulnerable/non-vulnerable), the prediction for the prioritization of vulnerable files requires a continuous value (number of vulnerabilities per file). The case study thus becomes a regression problem, and supervised learning algorithms should be chosen accordingly (i.e. one needs regression algorithms) [23].

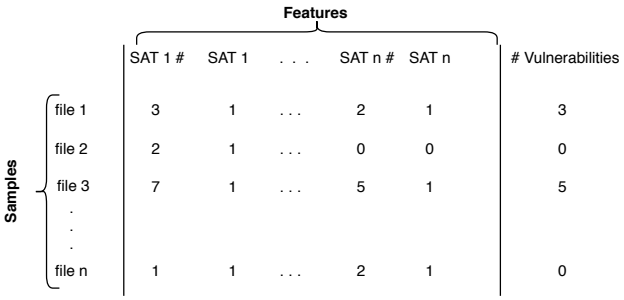

Fig. 7. Representation of the file dataset with the actual vulnerability number

Three regression algorithms were studied: *linear regression* (uses the ordinary least squares method), *decision trees regression* (uses the DT classification algorithm as base for this regressor), and *Lasso* (model that estimates sparse coefficients). Due to time constraints, it was not possible to thoroughly explore and tune all the meta-learner hyperparameters. Thus, an ad-hoc approach based on the default values of Scikit-learn is used [34]. Nevertheless, the obtained results allow prioritizing the files by the potential presence of vulnerabilities per file, and hence show that this is a promising approach.

The dataset is divided into training and testing sets (5-fold CV). To evaluate the results, three performance metrics were used: *a)* number of actual vulnerabilities in a Top-N set of files; *b)* proportion of vulnerabilities estimated in a Top-N set of files compared to the real number of vulnerabilities known; and *c)* overlap proportion of files in the two sets (Top-N prioritized by the regression algorithm and Top-N prioritized by the number of vulnerabilities).

C. Results and Discussion

Using the regression models, we calculated a prediction of the number of vulnerabilities for each file. The predicted value allows sorting the files by the number of predicted vulnerabilities. The results presented in this section are only for *linear regression*, as it consistently performed better than the other two approaches (*decision trees* and *Lasso*). Complete results can be found online (see footnote 1). The coefficient of determination r^2 is 0.8253 for *linear regression*.

Table V shows the estimates for both SQLi and XSS using linear regression, i.e. actual number of vulnerabilities in the Top-50 files (sorted by the number of vulnerabilities). This corresponds to metrics *a* and *b* presented above. When analyzing the Top-50 files, we can observe that they contain a large portion of the total set of true vulnerabilities (58.06% for SQLi and 35.40% for XSS). This largely overlaps the Top-50 files when ranked by the actual number of vulnerabilities (64.03% for SQLi and 40.93% for XSS). Note that the number of files in the analysis (50 in our case) can be adjusted according to the needs of the project at hands.

A way to understand how the number of detected vulnerabilities increases is to analyze the progression of vulnerabilities when the number of top files (*N*) is enlarged. Fig. 8 shows the proportion of vulnerabilities in Top-N files as estimated by the regression algorithm in comparison to the Top-N list sorted by

TABLE V
VULNERABILITIES IDENTIFIED BY THE TOP-50 FILES USING LINEAR REGRESSION AND ACTUAL NUMBER OF VULNERABILITIES

	Vulnerability Count (%)	
	Regression	**Actual**
SQLi	389 (58.06%)	429 (64.03%)
XSS	1,745 (35.40%)	2,018 (40.93%)

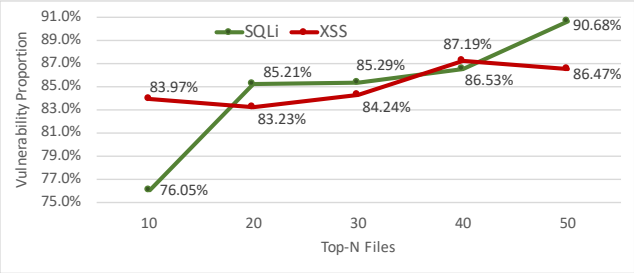

Fig. 8. Proportion of vulnerabilities detected by the Top-N files

the number of true vulnerabilities, varying the N from 10 to 50 files, in increments of 10.

The results are quite good, as the proportion of vulnerabilities detected is always above 75.0%. For example, for the Top-10, the proportion is 76.05% for SQLi and 83.97% for XSS vulnerabilities. As it can be noticed, the proportion always increases for SQLi, but the same does not happen for XSS. For instance, the proportion decreases from the Top-10 to the Top-20 files. This is related to the fact that the selected files in that range (Top-11 to Top-20) have less predicted vulnerabilities than real values. Although the proportion decreases, the number of vulnerabilities increases as more files are added.

Another analysis that can be made is the comparison of the ordered lists of files. This can be done by analyzing the list created by the regression techniques with the list ranked by the actual number of vulnerabilities. Fig. 9 represents the overlap of files of the prioritized lists created by the regression techniques and the actual number of vulnerabilities. Although the order may not be the same, what is important is whether the two lists have a large overlap of files (metric c presented in the previous section). As can be seen in Fig. 9, the two lists have a large overlap, where the only exception is for the Top-10 files for SQLi, where the overlap is only 50.0%. For all the other cases, the overlap is larger, varying from 60.0% to 76.0%. This is an interesting outcome, as it shows that the regression-based list largely overlaps the top set of files when sorted by the number of actual vulnerabilities.

As shown, our prioritization approach provides a very good starting point to identify the files that should be analyzed first. Nevertheless, this should be combined with the output of the vulnerability detection using ML classification algorithms (see Section III).

V. THREATS TO VALIDITY

The work presented in this paper is an exploratory study on the use of ML algorithms to improve vulnerability detection.

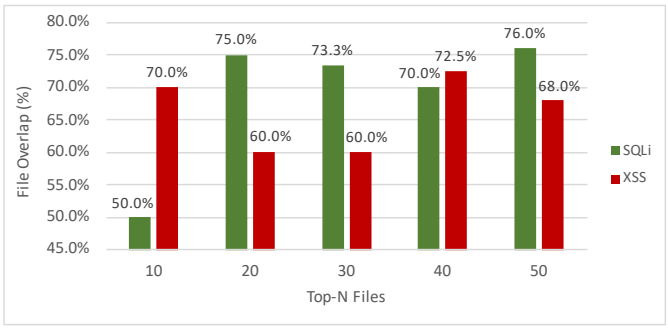

Fig. 9. Proportion of files detected compared with the Top-N files

Although the algorithms used are some of the most relevant in the state-of-the-art, other algorithms may also present good results for the same problem.

The dataset was created from the SATs of various alerts. Only a small number of known vulnerabilities in the plugins were not detected by those SATs (some vulnerabilities in the original dataset from *Nunes et al.* [19] were either manually detected or they were listed in the WPVD). Nonetheless, there may be unknown vulnerabilities in the source code, thus we cannot state that the dataset contains all vulnerable SSs.

The alerts were manually analyzed and classified by experts either as vulnerable or non-vulnerable. Although they have experience in security and in SQLi and XSS vulnerabilities, they may have committed mistake when labeling the SSs, which may influence the results of this study.

The original dataset contained vulnerabilities that are not identifiable by any of the SATs. Consequently, the ML algorithms do not have a way to predict these SSs as vulnerable. Hence, other features (either other SATs or another source of vulnerability information) can be used when running the ML algorithms.

The file prioritization does not consider how critical a vulnerability or the likelihood of being exploited. Although the approach highlights the files that potentially contain many vulnerabilities, those may not be the ones that lead to more damage when exploited.

VI. CONCLUSION AND FUTURE WORK

Software vulnerabilities are a relevant problem in software as their consequences may involve data and financial losses. Thus, detecting them before releasing the software to production is of utmost importance. Approaches that combine the output of several SATs have good results, but they suffer from a key problem: a high number of FPs.

This exploratory study shows that is possible to decrease the number of FPs for SQLi vulnerabilities in a dataset that contains alerts from different SATs on WordPress plugins. Good results are obtained for XSS vulnerabilities, but they are equivalent to the *1ooN* heuristic. The study also shows that using linear regression based on the output of SATs allows prioritizing the files with potentially more SQLi and XSS vulnerabilities. This can be used as a starting point of analysis by the development teams.

For future work, the same ML algorithms need to be applied considering other vulnerability detection techniques, including software metrics. Additionally, dynamic techniques can be used to either confirm an alert as a vulnerability or reject it as a FP. The prioritization mechanism also needs to be improved using alternative approaches. Finally, the same techniques need to be applied in other datasets, either of different project size, different programming languages, or of other vulnerability types. This will allow to propose a systematic methodology, in addition to confirm it for projects with different characteristics.

ACKNOWLEDGMENT

This work was partially funded by FCT grant no. SFRH/BD/140221/2018, project **ATMOSPHERE**, funded by the European Commission under the Cooperation Programme, H2020 grant agreement no. 777154, and project **METRICS**, funded by the FCT – agreement no `POCI-01-0145-FEDER-032504`. We would like to thank Paulo Nunes for the support and for the datasets.

REFERENCES

[1] S. Moore and E. Keen, "Gartner Forecasts Worldwide Information Security Spending to Exceed $124 Billion in 2019," https://www.gartner.com/en/newsroom/press-releases/2018-08-15-gartner-forecasts-worldwide-information-security-spending-to-exceed-124-billion-in-2019, 2018, accessed: 2019-06-03.

[2] P. Voigt and A. v. d. Bussche, *The EU General Data Protection Regulation (GDPR): A Practical Guide*, 1st ed. Springer Publishing Company, Incorporated, 2017.

[3] I. Muscat, "Cyber Threats vs Vulnerabilities vs Risks," https://www.acunetix.com/blog/articles/cyber-threats-vulnerabilities-risks/, 2017, accessed: 2019-06-20.

[4] B. Liu, L. Shi, Z. Cai, and M. Li, "Software Vulnerability Discovery Techniques: A Survey," in *2012 Fourth International Conference on Multimedia Information Networking and Security*, Nov 2012, pp. 152–156.

[5] N. Mead, E. Hough, and T. S. II, "Security Quality Requirements Engineering Technical Report," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2005-TR-009, 2005. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7657

[6] K. Turpin, "OWASP Secure Coding Practices - Quick Reference Guide," https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf, 2010, accessed: 2019-06-20.

[7] B. Chess and J. West, *Secure Programming with Static Analysis*, 1st ed. Addison-Wesley Professional, 2007.

[8] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security Privacy*, vol. 2, no. 6, pp. 76–79, Nov 2004.

[9] A. Algaith, P. J. C. Nunes, F. D. R. S. Jose, I. Gashi, and M. Vieira, "Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools," *2018 14th European Dependable Computing Conference (EDCC)*, pp. 57–64, 2018.

[10] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," *CoRR*, vol. abs/1807.04320, 2018.

[11] J. Walden, J. Stuckman, and R. Scandariato, "Predicting Vulnerable Components: Software Metrics vs Text Mining," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, Nov 2014, pp. 23–33.

[12] H. Alves, B. Fonseca, and N. Antunes, "Experimenting Machine Learning Techniques to Predict Vulnerabilities," in *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, Oct 2016, pp. 151–156.

[13] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira, "On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study," in *2017 13th European Dependable Computing Conference (EDCC)*, Sep. 2017, pp. 121–128.

[14] A. van der Stock, B. Glas, N. Smithline, and T. Gigler, "OWASP Top 10 - 2017 - The Ten Most Critical Web Application Security Risks," https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf, 2017, accessed: 2019-05-04.

[15] P. Louridas, "Static code analysis," *IEEE Software*, vol. 23, no. 4, pp. 58–61, July 2006.

[16] A. Austin, C. Holmgreen, and L. Williams, "A comparison of the efficiency and effectiveness of vulnerability discovery techniques," *Information and Software Technology*, vol. 55, no. 7, pp. 1279 – 1288, 2013.

[17] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, Sep. 2008.

[18] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic Creation of SQL Injection and Cross-site Scripting Attacks," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 199–209.

[19] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira, "Benchmarking Static Analysis Tools for Web Security," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1159–1175, Sep. 2018.

[20] N. Antunes and M. Vieira, "Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services," in *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, Nov 2009, pp. 301–306.

[21] R. Scandariato, J. Walden, and W. Joosen, "Static analysis versus penetration testing: A controlled experiment," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2013, pp. 451–460.

[22] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *2011 International Symposium on Empirical Software Engineering and Measurement*, Sep. 2011, pp. 97–106.

[23] E. Alpaydin, *Introduction to Machine Learning*. The MIT Press, 2014.

[24] J. Marques de Sá, *Pattern Recognition*, 1st ed. Springer-Verlag Berlin Heidelberg, 2001.

[25] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, June 1994.

[26] WPScan, "WPScan Vulnerability Database," https://wpvulndb.com, 2014, accessed: 2019-06-13.

[27] M. Trego, "Top 10 Programming Languages Used in Web Development," https://blog.stoneriverelearning.com/top-10-programming-languages-used-in-web-development/, 2017, accessed: 2019-08-15.

[28] P. J. C. Nunes, J. Fonseca, and M. Vieira, "phpSAFE: A Security Analysis Tool for OOP Web Application Plugins," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015, pp. 299–306.

[29] J. Dahse and T. Holz, "Simulation of Built-in PHP Features for Precise Static Code Analysis," in *In Symposium on Network and Distributed System Security (NDSS)*, 01 2014, pp. 23–26.

[30] I. Medeiros, N. F. Neves, and M. Correia, "Automatic Detection and Correction of Web Application Vulnerabilities Using Data Mining to Predict False Positives," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14. New York, NY, USA: ACM, 2014, pp. 63–74.

[31] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting Web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S P'06)*, May 2006, pp. 6 pp.–263.

[32] D. Hauzar and J. Kofron, "Framework for Static Analysis of PHP Applications," in *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), J. T. Boyland, Ed., vol. 37. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 689–711.

[33] J. R. Campos, M. Vieira, and E. Costa, "Propheticus: Machine Learning Framework for the Development of Predictive Models for Reliable and Secure Software," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Oct 2019.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.