Discovering Software Vulnerabilities Using Data-flow Analysis and Machine Learning

Discovering software vulnerabilities using data-flow analysis and machine learning

Jorrit Kronjee
Open University of the Netherlands
Heerlen, The Netherlands
jorrit@wafel.org

Arjen Hommersom
Open University of the Netherlands
Heerlen, The Netherlands
Radboud University
Nijmegen, The Netherlands
Arjen.Hommersom@ou.nl

Harald Vranken
Open University of the Netherlands
Heerlen, The Netherlands
Radboud University
Nijmegen, The Netherlands
Harald.Vranken@ou.nl

ABSTRACT

We present a novel method for static analysis in which we combine data-flow analysis with machine learning to detect SQL injection (SQLi) and Cross-Site Scripting (XSS) vulnerabilities in PHP applications. We assembled a dataset from the National Vulnerability Database and the SAMATE project, containing vulnerable PHP code samples and their patched versions in which the vulnerability is solved. We extracted features from the code samples by applying data-flow analysis techniques, including reaching definitions analysis, taint analysis, and reaching constants analysis. We used these features in machine learning to train various probabilistic classifiers. To demonstrate the effectiveness of our approach, we built a tool called *WIRECAML*, and compared our tool to other tools for vulnerability detection in PHP code. Our tool performed best for detecting both SQLi and XSS vulnerabilities. We also tried our approach on a number of open-source software applications, and found a previously unknown vulnerability in a photo-sharing web application.

CCS CONCEPTS

• **Security and privacy** → **Vulnerability scanners**; **Software security engineering**; **Web application security**; • **Computing methodologies** → **Supervised learning by classification**; • **Software and its engineering** → **Automated static analysis**;

KEYWORDS

Software security, vulnerability detection, static code analysis, machine learning, data-flow analysis

ACM Reference Format:
Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. 2018. Discovering software vulnerabilities using data-flow analysis and machine learning. In *ARES 2018: International Conference on Availability, Reliability and Security, August 27–30, 2018, Hamburg, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3230833.3230856

1 INTRODUCTION

Despite the increasing efforts for improving software security, vulnerabilities in software code are still of major concern. For instance, 5000 to 8000 newly discovered vulnerabilities have been reported yearly since 2012 in the National Vulnerability Database (NVD), and in 2017 more than 14000 vulnerabilities were reported [18]. Vulnerability discovery is often a tedious task which requires intimate knowledge of the system, the programming language, and possible attack scenarios. The difficulty of these tasks creates a strong demand for new methods that facilitate vulnerability discovery.

Various methods have been developed for discovering vulnerabilities in software code, which can be categorized as manual inspection, static (code) analysis, and dynamic analysis. Each of these methods has its limitations. As source code projects grow, manual inspection quickly becomes unfeasible.

Static analysis is performed by analysing the program code without running it. The main advantage of static analysis is that all paths in the code can be considered. Typically, a predefined set of rules is used to find vulnerabilities such as the use of insecure library functions, buffer overflows, or insufficient data validation. An obvious limitation is that only the vulnerabilities covered by the rule set can be discovered. The first static analysis tools appeared early 2000 (such as Flawfinder, ITS4, and RATS) [4]. They simply checked the usage of potentially dangerous functions such as `strcpy()`. Since then, many more advanced tools have been developed. Most tools employ taint analysis, which checks the usage of variables that originate from input that can be modified (directly or indirectly) by the user. Dynamic languages have been a challenge for static analysis, since properties such as run-time source inclusion, usage of `eval` functions, object runtime alteration, and reflection can only be fully evaluated at run-time.

Dynamic analysis is performed by testing the program code during run-time and can be applied with or without the availability of source code. Dynamic analysis can be done by vulnerability scanners that use a defined set of malicious patterns. Although they are popular for penetration testing, they cannot guarantee complete code coverage since the code is tested without knowledge of the code structure. Furthermore, they are limited to the set of malicious test patterns applied. Fuzzers improve this to some extent by generating semi-random test patterns, and some fuzzers add instrumentation to the code for discovering code branches [13].

In this paper, we combine data-flow analysis with machine learning to create a novel method for static analysis to identify vulnerabilities. We created a dataset containing PHP source code files that each have a SQL injection (SQLi) or Cross-Site Scripting (XSS) vulnerability, as well as the patched versions of the files in which the vulnerability is solved. Next, we generated control-flow graphs (CFGs) for the code in each file. We extracted features from these CFGs that relate to data flow, and we applied these features in machine learning to train various probabilistic classifiers. Given an input, a probabilistic classifier outputs not only the most likely class that the input belongs to, but also the probability distribution considering all classes. In our case, the outputs are the probabilities that the code contains either a SQLi or XSS vulnerability.

We provide the following contributions: (1) we show that relevant features can be extracted from CFGs; (2) we show that these features can be used successfully to train a probabilistic classifier for detecting SQLi and XSS vulnerabilities with high accuracy; (3) we compare the performance of various probabilistic classifiers with these features; (4) we compare the detection accuracy of our approach against other static code analysis tools; (5) we demonstrate that our approach can be used to find vulnerabilities in open-source PHP software code that have not been discovered before. We implemented our approach in Python and created tooling named *WIRECAML*, which is available as open-source software under the MIT license [11].

The remainder of the paper is organised as follows: We first describe related work on the usage of machine learning for vulnerability detection in Section 2. Next, we present how we extract relevant features from CFGs in Section 3. We outline our dataset, experimental setup, and the performance of the classifiers in Section 4. In Section 5 we compare the performance of our tool against other tools, and in Section 6 we apply our tool for detecting vulnerabilities in a number of open-source software applications. We discuss limitations of our approach and directions for future work in Section 7. We conclude the paper with Section 8.

2 RELATED WORK

A recent survey by Ghaffarian et al. [8] split work in the area of machine learning and vulnerability discovery into three main categories: vulnerability prediction based on software metrics, anomaly detection, and vulnerable code pattern recognition. Only very modest results have been achieved so far with vulnerability prediction based on software metrics [6, 32, 35, 37]. Vulnerability discovery by anomaly detection is only effectively applicable for mature software systems, and is unable to distinguish vulnerabilities from defects [42]. Our work can be categorised as vulnerable code pattern recognition, which we consider as the most promising category.

Vulnerable code pattern recognition analyses and extracts features from program source code, striving to extract models and patterns of vulnerable code. The common theme is to gather a large dataset of vulnerable samples, process them to extract feature vectors from each sample, and utilise machine learning algorithms to automatically learn a pattern recognition model for software vulnerabilities. To this end, different approaches are used to process and extract features from program source code, such as code parsing, static data-flow and control-flow analysis, dynamic analysis, and text mining.

Yamaguchi et al. proposed a method for assisted discovery of vulnerabilities in source code by introducing the concept of 'vulnerability extrapolation' [39, 40]. Their method proceeds by extracting abstract syntax trees from the code and determining structural patterns in these trees, such that each function in the code can be described as a mixture of these patterns. This representation enables the deconstruction of a known vulnerability into a graph query and extrapolates it to a code base, such that functions potentially suffering from the same flaw can be suggested to the analyst. In later research, these researchers proposed a new graph representation named 'code property graph' for modelling and discovering vulnerabilities by means of graph traversals [38, 41]. The code property graph merges concepts of classic program analysis (abstract syntax trees, control-flow graphs, and program dependence graphs) into a joint data structure. This data structure is then stored in a graph database allowing an analyst to construct template queries to find new instances of a known vulnerability. Although this is a novel approach to assist a security analyst, it requires a fair amount of knowledge of both the application that is investigated and the query language. Moreover, this research was limited to C and C++.

Shar and Tan investigated PHP and proposed a set of 20 static code attributes based on data-flow analysis that can be used to predict program statements that are vulnerable to SQLi and XSS attacks [29, 30]. These 20 static code attributes reflect different data-flow aspects of a code segment, such as the number of statements that input data from various sources (e.g., HTTP requests, files, databases), the type of input data, the number of different output sink statements (e.g., database queries, HTML outputs), and the number of different input validation and sanitisation statements. To evaluate the effectiveness of the attributes, the authors developed a tool called *PhpMiner* and performed experiments on eight open source web applications written in PHP. The authors also compared the performance of *PhpMiner* against *Pixy*. On average, *Pixy* discovered more vulnerabilities but also produced many more false positives compared to *PhpMiner*. (We compared *Pixy* with our tooling, see Section 5.)

In later research, Shar et al. predicted vulnerabilities using hybrid code attributes [31]. Dynamic analysis was incorporated into static analysis to improve the classification accuracy. Shar et al. further extended their work by adding Remote Code Execution and File Inclusion to their vulnerability scope and employing a technique called 'static backward program slicing' in order to extract different execution paths from program slices [28]. They also modified their feature set to include 10 static and 22 dynamic attributes and used a semi-supervised approach alongside the supervised approach for vulnerability prediction, where the semi-supervised approach can be used when there is a shortage of labelled training data. These were then implemented in a new version of *PhpMiner*. (We tried to obtain a copy of *PhpMiner* and its dataset for comparison, unfortunately without success.)

Medeiros et al. [12] asserted that taint analysis could be an effective mechanism for vulnerability discovery if machine learning is used to remedy the high false-positives rate. The proposed approach was implemented as a tool named Web Application Protection (*WAP*), consisting of several steps: first, taint flow analysis is performed on PHP source code to identify possible XSS, SQLi, file inclusion, and OS command injection vulnerabilities. Second, the trained Logistic Regression classifier was used to exclude false-positive reports based on 14 manually selected attributes. Third, a fixed set of code templates was used to correct the detected vulnerable sinks in the source code. For evaluation, the authors compared the results of *WAP* against *Pixy* and *PhpMiner*. The reported results indicate that *WAP* performs significantly better than the other tools.

3 EXTRACTING FEATURES

As a first step in our approach, we extract features. We parse the PHP source code files in our dataset and convert them into abstract syntax trees (ASTs) using the *phply* parser [21]. Next, we derive CFGs from the ASTs, from which we extract features. Although our research focuses on PHP, we select language-neutral features as much as possible, so that our approach can be applied to other dynamic languages in later research. A CFG represents all paths that might be traversed through program code during its execution [1]. It is a directed graph where the nodes represent basic blocks and edges represent possible transfer of control flow from one basic block to another. CFGs are commonly used in data-flow analysis. We apply reaching definitions analysis, taint analysis, and reaching constants analysis to extract features from the CFGs, as described in the following subsections.

3.1 Reaching definitions analysis

Reaching definitions analysis is a data-flow analysis technique which statically determines which definitions may reach a given point in the code [17]. A definition of a variable x is a statement that assigns, or may assign, a value to x. A definition `d` is said to reach a point `p` if there exists a path from `d` to `p` such that `d` is not *killed*, i.e. not overwritten by another definition of the same variable, along that path. For each basic block `b` in a CFG, four sets are defined: `IN[b]` represents all incoming definitions from preceding blocks; `GEN[b]` represents the definitions made inside `b`; `KILL[b]` represents the definitions in all other basic blocks of the CFG that are killed by the definitions in `b`; `OUT[b]` represents the definitions at the output of block `b`, where `OUT[b] = GEN[b]` ∪ `(IN[b] - KILL[b])`. Once we have constructed the reaching definitions, we determine the use-definition chains (UD chains) for each definition. A UD chain consists of a use of a variable, and all the definitions of that variable that can reach that use without any other intervening definitions. SQLi and XSS vulnerabilities are typically due to the use of potentially vulnerable functions and a lack of sanitisation. We use the UD chain for each line in the program code to determine which functions may have been used in the paths to that line. We consider the presence of functions as features.

Figure 1 shows the CFG of an example program that accepts user input, creates a SQL query with the supplied input,

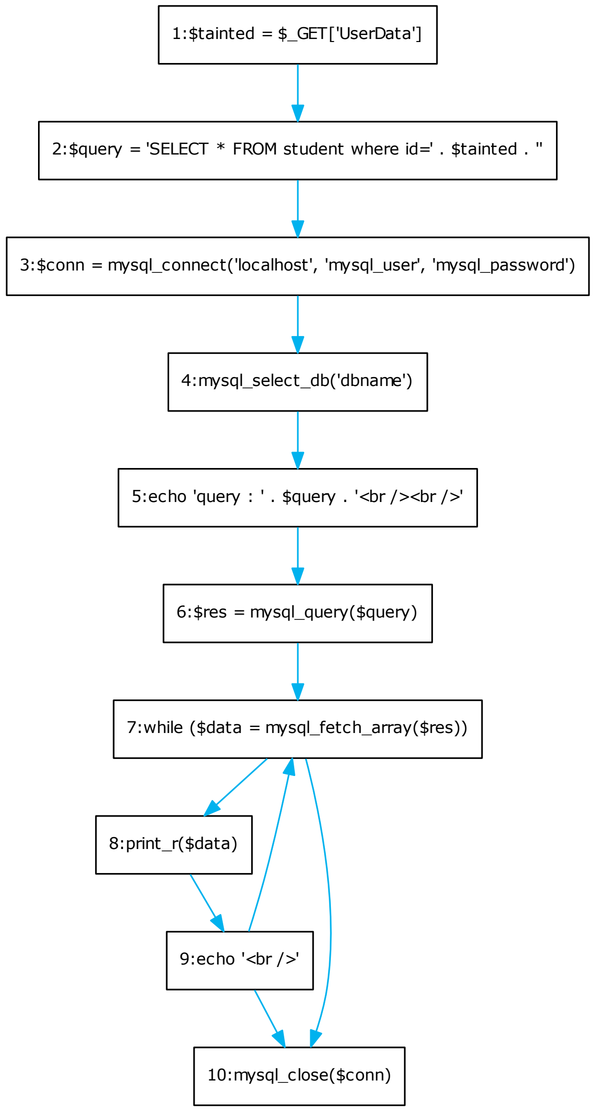
1:$tainted = $_GET['UserData']
2:$query = 'SELECT * FROM student where id=' . $tainted . ''
3:$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password')
4:mysql_select_db('dbname')
5:echo 'query : ' . $query . '<br /><br />'
6:$res = mysql_query($query)
7:while ($data = mysql_fetch_array($res))
8:print_r($data)
9:echo '<br />'
10:mysql_close($conn)

Figure 1: Example of a control-flow graph

line	echo	mysql _close	mysql _connect	mysql _fetch _array	mysql _query	mysql _select _db	print _r	**vuln.**
1	0	0	0	0	0	0	0	**0**
2	0	0	0	0	0	0	0	**0**
3	0	0	1	0	0	0	0	**0**
4	0	0	0	0	0	1	0	**0**
5	1	0	0	0	0	0	0	**0**
6	0	0	0	0	1	0	0	**1**
7	0	0	0	1	1	0	0	**1**
8	0	0	0	1	1	0	1	**1**
9	1	0	0	0	0	0	0	**0**
10	0	1	1	0	0	0	0	**0**

Table 1: Dataset derived from the CFG of Figure 1

executes the query, and displays the results. The program contains a SQLi vulnerability as the user input is not sanitised. The injected SQL query is executed on line 6, while the results of the query are used on lines 7 and 8. We therefore consider lines 6, 7, and 8 as vulnerable.

Using UD chains, we create Table 1 which shows the functions each line depends upon. For instance, line 7 uses `mysql_fetch_array` with parameter `$res`, which is defined on line 6 by function `mysql_query`. The last column indicates whether a line is marked as vulnerable. This table is an appropriate input for our probabilistic classifiers, where the vulnerable column is used as the class and the other columns (except the first column) are used as the features.

3.2 Taint analysis

Tainted data is data that can be modified (either directly or indirectly) by potentially malicious users, and thus can cause security problems at vulnerable points in the program (called sensitive sinks). Tainted data may enter the program through unsafe channels, and can spread across the program via assignments and similar constructs.

We assume that a variable is untainted when its type is *float*, *int*, *double*, or *bool*. This also applies when a variable is casted to these types, by functions such as `floatval` or `intval`. Variables that are not untainted are considered tainted. Variables that are composites of other variables follow the rules shown in Table 2.

By (re)using the reaching definitions (see Section 3.1), we can identify for each line whether it contains tainted data. When a line contains multiple variables, the rules in Table 2 are followed. Simply put, if at least one variable is tainted, the whole line is considered tainted. We use the taintedness of each line as input feature for our probabilistic classifiers. In this analysis, we do not take possible sanitisation of the tainted variables into account.

3.3 Reaching constants analysis

Tainted data can be sanitised, which removes harmful properties and hence transforms tainted data into untainted data. Vulnerabilities such as SQLi and XSS can be avoided by proper input sanitisation. Sanitisation can be done in various ways in the program code, using predefined filters or dedicated code. In PHP the function `filter_var(`*var*`,` *filtername*`,` *options*`)` filters the variable specified in the first argument *var* with a filter specified by the constant *filtername*. Such filters can be used for sanitising. For instance, the filter `FILTER_SANITIZE_NUMBER_INT` removes all characters except digits, plus, and minus sign, while `FILTER_SANITIZE_STRING` strips all tags and optionally strips or encodes special characters of a string.

$a	$b	Result
untainted	untainted	untainted
untainted	tainted	tainted
tainted	untainted	tainted
tainted	tainted	tainted

Table 2: Composing tainted variables $a and $b

Since identifying all possible sanitisation filters is infeasible and hard to maintain, we also include all constants as features using the same method as outlined in Section 3.1. In this way, the method could straightforwardly be applied to other dynamic languages where custom sanitisation functions with filters as parameters are used.

4 TRAINING THE CLASSIFIERS

In this section, we will first describe the dataset that we used for training and evaluation. Then, we evaluate the performance of five different probabilistic classifiers for this problem. Finally, we study the impact of the three different types of features that were discussed in the previous section on the predictive performance.

4.1 Dataset

We assembled a dataset from the NVD [18] and the SAMATE dataset [26].

The NVD is a vulnerability database maintained by NIST, containing real-world code snippets of vulnerable code. It is based upon the Common Vulnerabilities and Exposures (CVE) [14] standard vulnerability dictionary. The vulnerabilities contained in the NVD are classified according to the Common Weakness Enumeration (CWE) [15]. We only extracted vulnerabilities from the NVD classified as CWE-79 and CWE-89, which relate to XSS and SQLi vulnerabilities respectively, and selected only those code snippets that are written in PHP. We used the XML feed as provided by the NVD to extract relevant information from the years 2002 (when NVD started) to 2016. The NVD unfortunately does not contain patches, but instead links to the source code repositories were the code snippets originate from. We manually followed up these links (if available) and looked for the patches. In the end, we were able to derive 28 code snippets containing SQLi vulnerabilities and 81 code snippets containing XSS vulnerabilities from the NVD, as well as the corresponding patches. In the vulnerable code snippets, we marked the lines that are removed by the patches as the vulnerable lines. We consider all other lines as non-vulnerable.

We extended our dataset with test cases from the NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project. This dataset includes synthetic PHP test cases for SQLi and XSS vulnerabilities that were generated by a tool [33, 34]. The test cases are separated into safe (non-vulnerable) and unsafe (vulnerable) categories. We derived 3904 unsafe and 6176 safe XSS test cases, and 912 unsafe and 8640 safe SQLi test cases. We also derived an XML file containing the file names and vulnerable line numbers.

We assembled our dataset by including all of the vulnerable code snippets from the NVD and SAMATE. Since the number of non-vulnerable code snippets was much larger, we only included a random sample of the non-vulnerable code snippets in our dataset (20% of non-vulnerable files for SQLi, and 7% of non-vulnerable files for XSS). This sampling also allowed us to reduce the size of the dataset, which prevented memory issues for our tooling. The proportion of vulnerable to non-vulnerable code lines in our sampled dataset is 1 to 80 for SQLi and 1 to 67 for XSS.

4.2 Experimental setup

We split our dataset into three parts: a training set, a tuning set, and a test set, where the split was 70%, 10% and 20% respectively. Table 3 shows the sizes of these sets. We used the training set to train our classifiers, the tuning set to tune their parameters, and the test set to evaluate our classifiers. As a performance criterion, we chose the area under the precision-recall curve (AUC-PR), which works well in cases where the dataset is highly skewed [5], such as in our dataset. Given a set of predictions from a classifier that outputs a probability, we can obtain a deterministic classifier by classifying an example as vulnerable if the classifier outputs a probability $p > c$ for the vulnerability class, where c is a particular threshold. Suppose tp is the number of true positives, fp is the number of false positives, and fn is the number of false negatives, then precision is defined as:

$$Prec = \frac{tp}{tp + fp}$$

and recall or true-positive rate is defined as:

$$Recall = \frac{tp}{tp + fn}$$

The precision-recall (PR) curve shows the trade-off between precision and recall by varying the threshold c.

Based on the extracted features, we trained five basic classifiers: decision tree, random forest, logistic regression, naive Bayes, and tree augmented naive Bayes (TAN). See e.g. [7] for an overview of these techniques. Naive Bayes classifiers and TANs model a joint probability distribution over the features and the class variable (vulnerable / not vulnerable), with different underlying assumptions. Decision trees and logistic regression model a conditional distribution for the class variable given the features. Finally, random forest is an ensemble learning method where predictions from multiple decision trees are combined to improve their individual performance.

	SQL		**XSS**	
	non-vuln.	vuln.	non-vuln.	vuln.
training	153,011	1,898	189,312	2,848
tuning	23,418	311	27,230	390
testing	45,523	568	5,7468	841
total	221,952	2,777	274,010	4,079

Table 3: Number of lines included in datasets

Classification method	**Vulnerability**	**AUC-PR**
Decision tree	SQLi	**0.88**
Random forest	SQLi	0.85
Logistic regression	SQLi	0.87
Naive Bayes	SQLi	0.64
TAN	SQLi	0.75
Decision tree	XSS	**0.82**
Random forest	XSS	**0.82**
Logistic regression	XSS	0.79
Naive Bayes	XSS	0.69
TAN	XSS	0.81

Table 4: Comparison of several probabilistic classifiers in terms of AUC-PR values

We used the tuning set to determine the best hyperparameters, which are the parameters of a model training algorithm and are set prior to the learning process. To optimise these hyperparameters we used grid search, which is an exhaustive search through the space of the hyperparameters. As we are trying to optimise our models for the AUC-PR value, we use the AUC-PR value on the tuning set as a performance metric. This was implemented using the Python *scikit-learn* [20] library which contains various probabilistic classifiers as well as tools for feature extraction and normalization, choosing features, model selection, and validation.

4.3 Classifier performance

Table 4 shows the performance of the different probabilistic classifiers for detecting SQLi and XSS vulnerabilities. The first observation is that the AUC-PR is generally better for the SQLi than for the XSS vulnerabilities. Considering that XSS attacks are more diverse (stored vs. reflected, multiple sinks), it is possible that the performance of the XSS classifiers was affected by this extra complexity. Secondly, the decision tree classifier outperforms the other methods for detecting SQLi vulnerabilities, although the difference with the random forest and logistic regression classifiers is small. For XSS, all methods show a similar performance, except for the naive Bayes classifier.

Since the decision tree classifier outperforms the other methods in these experiments and is simpler than the random forest, we selected this classifier for further evaluation, which is described in the remainder of this paper. The precision-recall graphs of this classifier are shown in Figure 2 and Figure 3 respectively for both types of vulnerabilities.

4.4 Feature performance

To get a better feeling for which features are most relevant for the performance, we have looked at the contribution of the

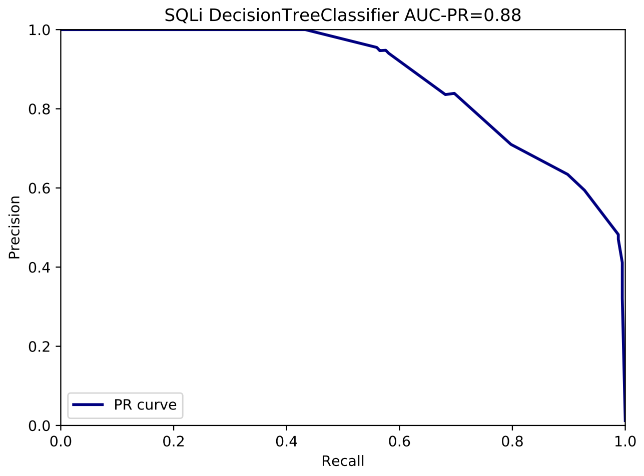

Figure 2: PR curve of decision tree classifier for SQLi

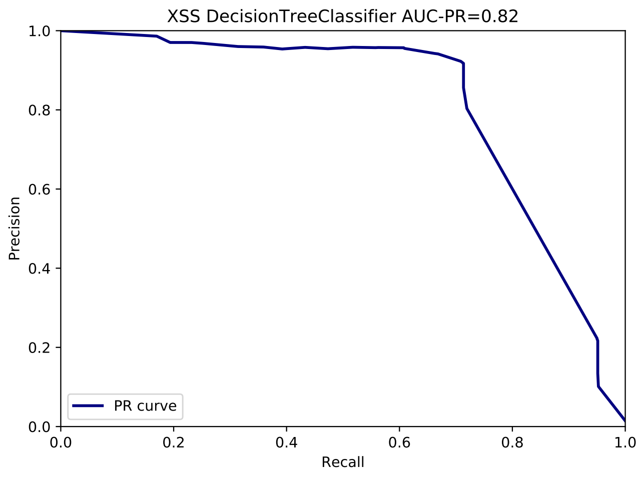

Figure 3: PR curve of decision tree classifier for XSS

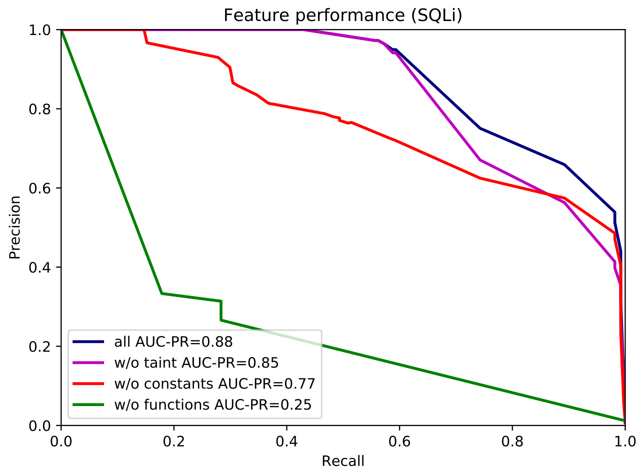

Figure 4: Contribution of feature types for SQLi

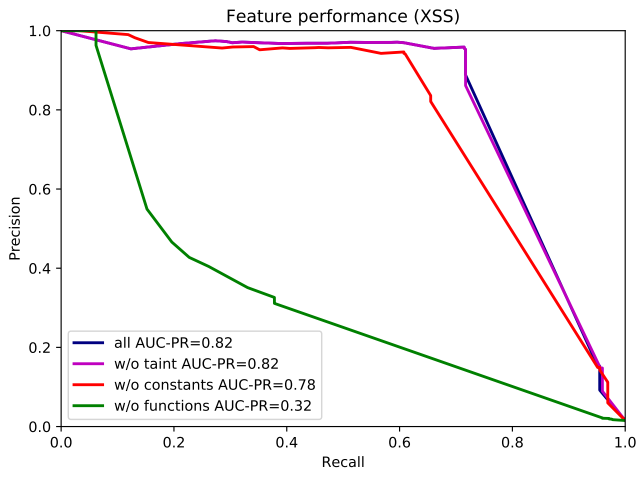

Figure 5: Contribution of feature types for XSS

three types of features that were introduced in the previous section, i.e., functions, taint analysis, and constants. To see their contribution, we have built models where we leave out one of these feature types to establish their contribution to the model's performance.

The results are shown in Figures 4 and 5. The PR-curve named 'all' is the combination of all feature types and is considered the baseline. Although all feature types seem to add to the model's performance, the taint feature seems to have little to negligible impact. This may have to do with the method used as our taint analysis does not take into account sanitisation functions, which means that most variables are tainted. It could also be because the information encoded by the taint feature is highly dependent on the other features.

5 COMPARISON TO OTHER TOOLS

Over the years various static code analysis tools have been created for PHP. Some of these tools are nothing more than a glorified 'grep', while others do some form of data-flow analysis. NIST maintains a curated list of source code security analysers [19]. In this section, we look at a comparison between our tool and these existing tools.

5.1 PHP static analysis tools

For our comparison, we selected candidate tools on completeness (is it able to find SQLi as well as XSS vulnerabilities?), implementation (can it do more than just searching for names of potentially vulnerable functions?), and output (does it show where the vulnerability is?). Based on these criteria, we selected four tools (*Pixy* [9], *RIPS* [24], *WAP* [12, 36], and *Yasca* [27]) to compare with our own tool.

Pixy is a static code analysis tool that scans PHP applications for security vulnerabilities. More precisely, flow-sensitive, interprocedural, and context-sensitive data-flow analysis are used to discover vulnerable points in a program. In addition, alias and literal analysis are employed to improve the correctness and precision of the results. *Pixy* was originally developed by Nenad Jovanonic. The current maintainer is Oliver Klee [10], although there has been no active development since 2014. We are using the version with commit ID 3f81106 for our comparison.

RIPS is the self-proclaimed most popular static code analysis tool to automatically detect vulnerabilities in PHP applications. By tokenising and parsing all source code files, *RIPS* is able to transform PHP source code into a program model and to detect sensitive sinks that can be tainted by user input during the program flow. Besides the structured output of found vulnerabilities, *RIPS* offers an integrated code audit framework. *RIPS* was originally written by Johannes Dahse and released as open-source software. In 2016, a new and rewritten version of *RIPS* was released as a commercial software product by RIPS Technologies [25] to overcome the technical limitations of the open-source version. We only evaluated the open-source version (version 0.55).

As explained in Section 2, *WAP* is a source code static analysis and data mining tool that detects and corrects input validation vulnerabilities in web applications written in PHP by semantically analysing source code. *WAP* was written by Ibéria Medeiros in 2014 and has been part of the OWASP project since 2015. We used version 2.1 for our comparison.

Yasca is an open source program written by Michael Scovetta which looks for security vulnerabilities, code quality, performance, and conformance to best practices in program source code. It leverages external open source programs, such as *FindBugs*, *PMD*, *JLint*, *JavaScript Lint*, *PHPLint*, *Cppcheck*, *ClamAV*, *Pixy*, and *RATS* to scan specific file types, and also contains many custom built-in scanners developed for *Yasca*. It can generate reports in HTML and JSON. We used version 3.0.5 for our comparison.

5.2 Preparation

Since none of these tools output probabilities, we will be using an F_1-score to compare the tools. F_1 can be calculated with

$$F_1 = 2 \cdot \frac{Prec \cdot Recall}{Prec + Recall}$$

The F_1-score is in fact a weighted average over both the vulnerable and the (much larger) non-vulnerable classes.

As our tool does output a probability of the vulnerable class, we determine the threshold that maximises the F_1-score of the classifier's predictions using the tuning set (i.e., beforehand). In other words, if Y represents the probability of the vulnerable class for the tuning set and we use $Y > c$ to determine the class label, we will find the threshold c that maximises the F_1-score. This threshold c will then be used to determine the F_1-score for the test set.

We only used the SAMATE dataset for our evaluation, as we noticed that none of the tools were able to cope with the larger NVD dataset. (Although the number of code snippets taken from the NVD is smaller, as indicated in Section 4.1, all dependent files have to be included as well to provide a complete code base.) After the transformation step, the SAMATE dataset consists of 28,268 non-vulnerable lines and 3,904 vulnerable lines for XSS, and 99,338 non-vulnerable lines and 2,736 vulnerable lines for SQLi. Our tool is trained for these experiments with a training set that is extracted from this dataset. The test set is not used during training.

Although all tools report a line number for a vulnerability, not every tool considers the vulnerability to occur in the same place. Some tools, such as *Pixy*, report the vulnerability at the sink, while other tools, such as *RIPS*, report all the lines between source and sink. To do a fair comparison, we decided to only compare files and ignore the line number. Since the SAMATE dataset only contains one vulnerability per file, this does not skew the results.

Some of the tools allow configuration (*Yasca*, for instance, has the possibility to disable or enable plug-ins). We assume that the default configuration generates the best results. We ignored any additional vulnerabilities that are reported other than SQLi and XSS in the results.

5.3 Results

Table 5 shows the results (weighted averages) for the SQLi dataset. It shows that our *WIRECAML* tool scored the best with an F_1-score of 0.94 with *WAP* and *RIPS* as runner-ups. When looking at the 'vulnerable' class results (Table 6) it becomes clear that both *RIPS* and *WAP* were struggling to find the vulnerable samples. *Yasca* scored extremely low in these results. Upon manual inspection we discovered that *Yasca* reported vulnerabilities in all samples, despite the fact that most of the samples were not vulnerable. This is why *Yasca* has an F_1 score of 0.00 for the non-vulnerable class (Table 7).

Table 8 shows the results (weighted averages) for the XSS dataset. As we have seen in Section 4.3, our classifier scores lower for XSS than for SQLi. This apparently also applies to other tools (see Table 8), and our *WIRECAML* tool still scores the best overall. We note that for the non-vulnerable class *RIPS* scores better (see Table 10). This is because *RIPS* was not able to detect any XSS vulnerabilities at all (hence, $F_1 = 0.00$ for the vulnerable class in Table 9).

6 EVALUATION IN PRACTICE

Given the good results with our dataset, we decided to see if our tool could detect vulnerabilities in some existing open-source software projects. In this section, we discuss the approach for the field experiment and a description of the vulnerability that we found.

6.1 Approach

We used the latest versions of the following projects: Joomla! (3.8.3), Kajona (6.2), moodle (3.4), MyBB (1.8.14), ownCloud (10.0.4), phpMyAdmin (4.7.7), Piwigo (2.9.2), Tiki Wiki CMS Groupware (17.1), Typo3 (9.0.0), WordPress (4.9.1), and Zen Cart (1.5.5f).

For each project we trained two decision tree classifiers: one for XSS vulnerabilities and one for SQLi vulnerabilities. The output of our tool was a CSV file containing the file name, line number, and probability for each sample. We used

	Precision	Recall	F_1-score
WIRECAML	**0.94**	**0.94**	**0.94**
Pixy	0.86	0.61	0.69
RIPS	0.83	0.80	0.82
WAP	0.83	0.84	0.83
Yasca	0.01	0.10	0.02

Table 5: Weighted averages (SQLi) (SAMATE SQLi dataset; $c = 0.97$)

	Precision	Recall	F_1-score
WIRECAML	**0.78**	**0.57**	**0.66**
Pixy	0.15	0.61	0.24
RIPS	0.14	0.21	0.17
WAP	0.11	0.09	0.10
Yasca	0.10	1.00	0.18

Table 6: F_1-scores for class 'vulnerable' (SQLi)

	Precision	Recall	F_1-score
WIRECAML	**0.95**	**0.98**	**0.97**
Pixy	0.94	0.61	0.74
RIPS	0.91	0.87	0.89
WAP	0.90	0.92	0.91
Yasca	0.00	0.00	0.00

Table 7: F_1-scores for class 'not vulnerable' (SQLi)

	Precision	Recall	F_1-score
WIRECAML	**0.79**	**0.71**	**0.71**
Pixy	0.61	0.61	0.61
RIPS	0.37	0.61	0.46
WAP	0.51	0.58	0.51
Yasca	0.24	0.25	0.24

Table 8: Weighted averages (XSS) (SAMATE XSS dataset; $c = 0.83$)

	Precision	Recall	F_1-score
WIRECAML	**0.58**	**0.93**	**0.71**
Pixy	0.50	0.51	0.51
RIPS	0.00	0.00	0.00
WAP	0.36	0.12	0.18
Yasca	0.00	0.00	0.00

Table 9: F_1-scores for class 'vulnerable' (XSS)

	Precision	Recall	F_1-score
WIRECAML	0.93	0.57	0.70
Pixy	0.69	0.68	0.68
RIPS	**0.61**	**1.00**	**0.76**
WAP	0.61	0.86	0.71
Yasca	0.39	0.41	0.40

Table 10: F_1-scores for class 'not vulnerable' (XSS)

this for manual inspection to confirm whether or not they are exploitable.

We ignored the samples where the probability that they are vulnerable is 0. This resulted in 487,548 possible positives. Although this is a lot, it is only 9% of the in total 5,183,277 code lines (non-comment lines of code (NCLOC) as measured by *phploc* [3]). During our inspection we started with analysing the samples most likely to be vulnerable. We limited our effort for manual inspection of the possible positives to 3 days, in which we were able to inspect 1,646 samples. Of these, we identified 28 samples as suspicious, originating from Joomla!, moodle, Piwigo, Tiki Wiki CMS Groupware, and Zen Cart. To verify if these were actual vulnerabilities, we installed these projects on a virtual machine running Apache and MySQL, and tried to find a suitable exploit. We limited our effort for this task to 2 days.

We quickly discovered that, even though at first glance input parameters seemed to be taken directly from the web browser, these input parameters were actually being sanitised by a preprocessing function. This was not obvious during manual inspection as the samples themselves were still using the `$_GET`, `$_POST`, and `$_REQUEST` arrays, but these were apparently being overwritten with sanitised versions. This sanitisation could not have been picked up by our tooling, as array operations are not supported yet. After further examination, we discovered that for most samples some form of sanitisation is in place making them false positives, except for one sample of Piwigo.

6.2 The Piwigo vulnerability

Piwigo also uses a generic sanitisation mechanism using PHP's `addslashes()` function, which returns a string with backslashes added before characters that need to be escaped. These characters are: single quote ('), double quote ("), backslash (\), and NUL (0x0). Although `addslashes()` is often used as a filter to prevent SQL injections, it does not provide complete protection against it.

In listing 1 we see how input parameter `$_POST['tags']` is processed without validation, even though it was sanitised by `addslashes()` earlier. It is assumed that `$_POST['tags']` contains an array of integers, and therefore it is not encapsulated within quotes in the SQL query string. This means that we can inject arbitrary SQL code using the POST parameter. For instance, we could send an array with a single element containing the string `-1) UNION (SELECT password FROM piwigo_users` for injection. This would create a result set consisting of all the hashed passwords of the users of the system. This result set would then be part of the page output as evident from line 18-22 in listing 1.

The security risk of this vulnerability is estimated as low with a CVSS[1] score of 3.8 because exploitation requires the attacker to be authenticated as administrator. We reported the vulnerability to the Piwigo team [16], who subsequently

[1]Common Vulnerability Scoring System (CVSS) is a free and open industry standard for assessing the severity of computer system security vulnerabilities.

```
if (isset($_POST['delete']) and isset($_POST['tags']))
{
  if (!isset($_POST['confirm_deletion']))
  {
    $page['errors'][] = l10n('You need to confirm
     deletion');
  }
  else
  {
    $query = '
SELECT name
  FROM '.TAGS_TABLE.'
  WHERE id IN ('.implode(',', $_POST['tags']).')
;';
    $tag_names = array_from_query($query, 'name');

    delete_tags($_POST['tags']);

    $page['infos'][] = l10n_dec(
      'The following tag was deleted', 'The %d
     following tags were deleted',
      count($tag_names)
      )
      .' : '.implode(', ', $tag_names);
  }
```

Listing 1: Piwigo tag deletion snippet

released a fix. We also tried a commercial vulnerability scanner, which did not reveal this vulnerability.

7 DISCUSSION AND FUTURE WORK

There are several directions for improving our current results. During the evaluation of our tool in practice, as outlined in Section 6, it became clear that not being able to recognise array elements in an array is a shortcoming of our tool. For PHP all input from a browser is passed on using predefined, global arrays (`$_GET`, `$_POST`, `$_REQUEST`, `$_COOKIE`, and `$_SERVER`). Not parsing these arrays properly generated many of the false positives we have seen. Jovanovic et al. described how array variables can be tracked [9].

Another area of improvement is regular expressions. Regular expressions are commonly used for either validation or sanitisation. Our classifiers flagged lines containing functions related to regular expressions, such as `preg_match()` and `preg_replace()`. However, in most cases, these regular expressions were formed correctly and could not be exploited. Testing these regular expressions automatically with a set of strings commonly used in SQLi and XSS vulnerabilities and using the results of these tests as extra features, could provide an interesting set of new features.

Up to now, we focused on the manual construction of relevant features. In machine learning, representation (feature) learning has shown promising results [2]. Comparing such automatically-learned features to the features derived from data-flow analysis would be very interesting.

Besides improving the results, we also aim to generalise our tool to other dynamic languages and to other vulnerabilities. The latter may for instance be achieved by including techniques such as Remote Code Execution and File Inclusion, as reported by Shar et al. [28].

During our research, we faced two major challenges: acquiring a suitable dataset with vulnerable and non-vulnerable cases, and evaluating our approach. A significant contribution of our work is the construction of a dataset for training classifiers, using data from the NVD and the SAMATE project. Obtaining patches for the NVD vulnerabilities was a tedious task. We also discovered some issues with the SAMATE dataset: some XSS samples were miscategorised as ‘unsafe’, and in the SQLi samples the sink was mislabelled. We reported both issues, including patches [22, 23] to fix them, to the author of the test case generator.

The second challenge is in the evaluation of our approach. Our dataset may contain a sampling bias as it only includes samples from open-source applications. Samples from commercial applications are missing, due to the fact that both the code as well as the vulnerability data of commercial applications are not publicly accessible. Assuming that all application code is available is a general limitation in our approach as an application may use third-party plug-ins and components for which the source code may be unavailable. There is also a bias in the comparison to other tools, as we limited ourselves to freely available open-source versions of SQLi/XSS vulnerability scanners. Considering that commercial tools use sophisticated rule sets, they are likely to score better than their free counterparts. We call upon vendors of commercial vulnerability detection to make statistics about their detection rate public.

8 CONCLUSIONS

In this paper, we combine machine learning and static code analysis to detect vulnerable code, focusing on code written in the dynamic programming language PHP. Our results show that using machine learning in combination with features extracted from control-flow graphs and abstract syntax trees is an effective approach for vulnerability detection in dynamic languages, in particular PHP applications.

We applied five basic machine learning methods (decision tree, random forest, logistic regression, naive Bayes, and tree augmented naive Bayes). We observed that they offer similar predictive performance on this problem, where decision tree performed slightly better and naive Bayes performed worse. This suggests that using the right features is more important than the machine learning algorithm that is employed. We show that the most important class features are the usage of certain functions in code snippets. Features related to constants (that may identify sanitisation filters) and taint analysis further boost the performance.

To demonstrate the effectiveness of our approach, we built a tool called *WIRECAML*, and compared our tool to other tools for vulnerability detection in PHP code. For both the SQLi and XSS datasets our tool scored best. Our tool has already shown impact in the real world, as within a limited amount of time, we found a vulnerability in the photo-sharing web application Piwigo. This vulnerability has been reported [16] and the Piwigo team has released a fix, which will be included in Piwigo 2.9.3.

REFERENCES

[1] Frances E. Allen. 1970. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*. ACM, 1–19. https://doi.org/10.1145/800028.808479

[2] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* 35, 8 (2013), 1798–1828.

[3] Sebastian Bergmann. 2018-02-09T09:54:03Z. Phploc: A Tool for Quickly Measuring the Size of a PHP Project. https://github.com/sebastianbergmann/phploc

[4] Brain Chess and Garry McGraw. 2004. Static Analysis for Security. *IEEE Security & Privacy* 2, 6 (2004), 76–79.

[5] Jesse Davis and Mark Goadrich. 2006. The Relationship between Precision-Recall and ROC Curves. In *Proceedings of the 23rd International Conference on Machine Learning*. ACM, 233–240.

[6] Maureen Doyle and James Walden. 2011. An Empirical Study of the Evolution of PHP Web Application Security. In *Third International Workshop On Security Measurements and Metrics (Metrisec)*. IEEE, 11–20.

[7] Peter Flach. 2012. *Machine learning: the art and science of algorithms that make sense of data*. Cambridge University Press.

[8] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *Comput. Surveys* 50, 4 (2017), 1–36. https://doi.org/10.1145/3092566

[9] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *IEEE Symposium on Security and Privacy (S&P'06)*. IEEE. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1624016

[10] Oliver Klee. 2012. Pixy Is a Scanner Static Code Analysis Tools That Scans PHP Applications for Security Vulnerabilities. https://github.com/oliverklee/pixy Accessed 2017-06-19.

[11] Jorrit Kronjee. 2018. WIRECAML: Weakness Identification Research Employing CFG Analysis and Machine Learning. https://github.com/jorkro/wirecaml

[12] Ibéria Medeiros, Nuno F Neves, and Miguel Correia. 2014. 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*. ACM, 63–74.

[13] Michal Zalewski. 2016. Technical "Whitepaper" for Afl-Fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt

[14] MITRE. 2016. CVE - Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/

[15] MITRE. 2017. CWE - Common Weakness Enumeration. https://cwe.mitre.org/

[16] MITRE. 2018. CVE - CVE-2018-6883. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6883

[17] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[18] National Vulnerability Database. 2018. NVD - Statistics Search. https://web.nvd.nist.gov/view/vuln/statistics

[19] NIST. 2017. Source Code Security Analyzers - SAMATE. https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html Accessed 2017-07-02.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[21] Stanisław Pitucha. 2010. Phply: PHP Parser Written in Python Using PLY. https://github.com/viraptor/phply Accessed 2017-09-10.

[22] Pull Request #1 2018. Stivalet/PHP-Vuln-Test-Suite-Generator. https://github.com/stivalet/PHP-Vuln-test-suite-generator/pull/1 Accessed 2018-03-17.

[23] Pull Request #2 2018. Stivalet/PHP-Vuln-Test-Suite-Generator. https://github.com/stivalet/PHP-Vuln-test-suite-generator/pull/2 Accessed 2018-03-17.

[24] RIPS 2018. Free PHP Security Scanner Using Static Code Analysis. http://rips-scanner.sourceforge.net/ Accessed 2018-03-28.

[25] RIPS Technologies 2017. RIPS - Static Code Analysis for PHP Security Vulnerabilities. https://www.ripstech.com/ Accessed 2017-07-01.

[26] SAMATE 2018. Software Assurance Metrics And Tool Evaluation Project Main Page. https://samate.nist.gov/Main_Page.html Accessed 2018-03-28.

[27] Michael Scovetta. 2017. http://www.scovetta.com/yasca.html Accessed 2017-05-17.

[28] Lwin Khin Shar, Lionel C. Briand, and Hee Beng Kuan Tan. 2015. Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning. *IEEE Transactions on Dependable and Secure Computing* 12, 6 (2015), 688–707. https://doi.org/10.1109/TDSC.2014.2373377

[29] Lwin Khin Shar and Hee Beng Kuan Tan. 2012. Predicting Common Web Application Vulnerabilities from Input Validation and Sanitization Code Patterns. In *Proceedings of the 27th IEEE/ACM Automated International Conference On Software Engineering (ASE)*. IEEE, 310–313.

[30] Lwin Khin Shar and Hee Beng Kuan Tan. 2013. Predicting SQL Injection and Cross Site Scripting Vulnerabilities through Mining Input Sanitization Patterns. *Information and Software Technology* 55, 10 (2013), 1767–1780. https://doi.org/10.1016/j.infsof.2013.04.002

[31] Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C. Briand. 2013. Mining SQL Injection and Cross Site Scripting Vulnerabilities Using Hybrid Program Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 642–651. http://dl.acm.org/citation.cfm?id=2486873

[32] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. 2011-11. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2011-11), 772–787. https://doi.org/10.1109/TSE.2010.81

[33] Bertrand Stivalet. 2014. PHP-Vuln-Test-Suite-Generator: PHP Synthetic Test Cases Generator. https://github.com/stivalet/PHP-Vuln-test-suite-generator Accessed 2016-04-12.

[34] Bertrand Stivalet and Elizabeth Fong. 2016. Large Scale Generation of Complex and Faulty PHP Test Cases. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 409–415. http://ieeexplore.ieee.org/abstract/document/7515499/

[35] James Walden, Jeff Stuckman, and Riccardo Scandariato. 2014. Predicting Vulnerable Components: Software Metrics vs Text Mining. In *IEEE 25th International Symposium On Software Reliability Engineering (ISSRE)*. IEEE, 23–33.

[36] WAP 2018. Web Application Protection. http://awap.sourceforge.net/ Accessed 2018-03-28.

[37] Dumidu Wijayasekara, Milos Manic, and Miles McQueen. 2014. Vulnerability Identification and Classification via Text Mining Bug Databases. In *IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 3612–3618. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7049035

[38] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *IEEE Symposium On Security and Privacy (SP)*. IEEE, 590–604.

[39] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies (WOOT'11)*. USENIX Association, 13–13.

[40] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 359–368.

[41] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 797–812.

[42] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 499–510.